

USENIX

conference

proceedings

5th USENIX Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA

April 16–18, 2008

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGCOMM
& ACM SIGOPS

Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation San Francisco, CA, USA April 16–18, 2008

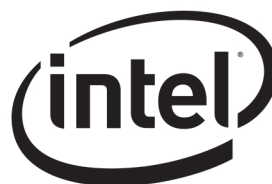
For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is \$40 for members and \$50 for nonmembers.
Outside the U.S.A. and Canada, please add \$20 per copy for postage (via air printed matter).

Thanks to Our Sponsors

Microsoft®
Research



Thanks to Our Media Sponsors

ACM Queue
Addison Wesley Professional/
Prentice Hall Professional
InfoSec News
ITToolbox
Linux Gazette

Linux Journal
Linux+DVD
Linux Pro Magazine
LXer
The Register
StorageNetworking.org

© 2008 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN-13: 978-1-931971-58-4

USENIX Association

**Proceedings of the
5th USENIX Symposium on
Networked Systems Design and
Implementation
(NSDI '08)**

**April 16–18, 2008
San Francisco, CA, USA**

Conference Organizers

Program Co-Chairs

Jon Crowcroft, *University of Cambridge*
Mike Dahlin, *University of Texas at Austin*

Program Committee

Paul Barham, *Microsoft Research*
Ken Birman, *Cornell University*
Miguel Castro, *Microsoft Research*
Jeff Chase, *Duke University*
Steve Gribble, *University of Washington*
Matthias Grossglauser, *Nokia Research Center/EPFL*
Krishna Gummadi, *Max Planck Institute for Software Systems*
Steven Hand, *University of Cambridge*
Brad Karp, *University College, London*
Dina Katabi, *Massachusetts Institute of Technology*
Eddie Kohler, *University of California, Los Angeles*
Sue Moon, *KAIST*
Robert Morris, *Massachusetts Institute of Technology*
Sylvia Ratnasamy, *Intel Research*
Luigi Rizzo, *ICIR*
Timothy Roscoe, *ETH Zürich*
Srinivasan Seshan, *Carnegie Mellon University*
Emin Gün Sirer, *Cornell University*
Amin Vahdat, *University of California, San Diego*
Arun Venkataramani, *University of Massachusetts Amherst*

Steering Committee

Thomas Anderson, *University of Washington*
Mike Jones, *Microsoft Research*
Greg Minshall
Robert Morris, *Massachusetts Institute of Technology*
Mike Schroeder, *Microsoft Research*
Amin Vahdat, *University of California, San Diego*
Ellie Young, *USENIX Association*

Poster Session Co-Chairs

Krishna Gummadi, *Max Planck Institute for Software Systems*
Arun Venkataramani, *University of Massachusetts Amherst*

Program Committee Light

David Culler, *University of California, Berkeley*
Peter Druschel, *Max Planck Institute for Software Systems*
Paul Francis, *Cornell University*
Deepak Ganesan, *University of Massachusetts Amherst*
Ramesh Govindan, *University of Southern California*
Mark Handley, *University College London*
John Hartman, *University of Arizona*
Joe Hellerstein, *Intel Research and University of California, Berkeley*
Rebecca Isaacs, *Microsoft Research, Cambridge*
Anne-Marie Kermarrec, *INRIA/IRISA*
Jinyang Li, *New York University*
Bryan Lyles, *Telcordia*
Petros Maniatis, *Intel Research, Berkeley*
Eugene Ng, *Rice University*
Adrian Perrig, *Carnegie Mellon University*
Jennifer Rexford, *Princeton University*
Dan Rubenstein, *Columbia University*
Stefan Savage, *University of California, San Diego*
Anees Shaikh, *IBM T.J. Watson Research Center*
Alex Snoeren, *University of California, San Diego*
Neil Spring, *University of Maryland*
Chandu Thekkath, *Microsoft Research, Silicon Valley*
Matt Welsh, *Harvard University*
David Wetherall, *University of Washington*
Ellen Zegura, *Georgia Institute of Technology*

USENIX Association Staff

External Reviewers

Aditya Akella
Periklis Akritidis
David Andersen
Katerina Argyraki
Suman Banerjee
Paul Barford
Andy Bavier
Ricardo Bianchini
Silas Boyd-Wickizer
Scott Brandt
Micah Brodsky
Ian Brown
Alvaro Cardenas
Meeyoung Cha
Ranveer Chandra
Byung-Gon Chun
Allen Clement
Manuel Costa
Landon Cox
David Culler
Frank Dabek
Colin Dixon
Peter Druschel
Dan Dumitriu
Daniel Ellard
Kevin Fall
Rodrigo Fonseca
Paul Francis
Michael Freedman
Timur Friedman
Deepak Ganesan
Roxana Geambasu
Johannes Gehrke
Christos Gkantsidis

Ramesh Govindan
Timothy Griffin
Ramakrishna Gummadi
Andreas Haeberlen
Mark Handley
Tim Harris
John Hartman
Joe Hellerstein
Tristan Henderson
Chi Ho
Ling Huang
Anthony Hylick
Rebecca Isaacs
Michael Kaminsky
Anne-Marie Kermarrec
Srinivasan Keshav
Changhoon Kim
Tadayoshi Kohno
Dejan Kostić
Ramakrishna Kotla
Christian Kreibich
Youngseok Lee
Kari Leppänen
Philip Levis
Harry Li
Jinyang Li
Jacob Lorch
Eric Yu-En Lu
Bryan Lyles
Anil Madhavapeddy
Harsha Madhyastha
Priya Mahadevan
Ratul Mahajan
Mike Mammarella

Petros Maniatis
Tudor Marian
David Mazières
Alan Mislove
Andrew Moore
Richard Mortier
Alex Moshchuk
Jayaram Mudigonda
Derek Murray
Suman Nath
Eugene Ng
David Oppenheimer
Stacy Patterson
Colin Perkins
Adrian Perrig
Aleksy Pesterev
Michael Piatek
Ansley Post
Lili Qiu
Pavlin Radoslavov
Venugopalan
Ramasubramanian
Charles Reis
Jennifer Rexford
Patrick Reynolds
Sean Rhea
Rodrigo Rodrigues
Antony Rowstron
Eric Rozner
Dan Rubenstein
Natasa Sarafijanovic-Djukic
Stefan Saroiu
Stefan Savage

Thomas Schmid
Anees Shaikh
Prashant Shenoy
Piyush Shivam
Thomas Silversen
Atul Singh
Emil Sit
Alex Snoeren
Yee Jun Song
Neil Spring
Peter Steenkiste
Jacob Strauss
Lakshminarayanan
Subramanian
Georgios
Theodorakopoulos
Eno Thereska
Robbert van Renesse
Steve VanDeBogart
Michael Walfish
Kevin Walsh
Andrew Warfield
Hakim Weatherspoon
Matt Welsh
David Wetherall
Alec Wolman
Bernard Wong
Alec Woo
Praveen Yalagandula
Alex Yip
Kenneth Yocum
Ellen Zegura

5th USENIX Symposium on Networked Systems Design and Implementation
April 16–18, 2008
San Francisco, CA, USA

Index of Authors	viii
Message from the Program Co-Chairs	ix

Wednesday, April 16

Trust

One Hop Reputations for Peer to Peer File Sharing Workloads	1
<i>Michael Piatek, Tomas Isdal, Arvind Krishnamurthy, and Thomas Anderson, University of Washington</i>	
Ostra: Leveraging Trust to Thwart Unwanted Communication	15
<i>Alan Mislove and Ansley Post, Max Planck Institute for Software Systems and Rice University; Peter Druschel and Krishna P. Gummadi, Max Planck Institute for Software Systems</i>	
Detecting In-Flight Page Changes with Web Tripwires	31
<i>Charles Reis, Steven D. Gribble, and Tadayoshi Kohno, University of Washington; Nicholas C. Weaver, International Computer Science Institute</i>	
Phalanx: Withstanding Multimillion-Node Botnets	45
<i>Colin Dixon, Thomas Anderson, and Arvind Krishnamurthy, University of Washington</i>	

Wireless

Harnessing Exposed Terminals in Wireless Networks	59
<i>Mythili Vutukuru, Kyle Jamieson, and Hari Balakrishnan, MIT Computer Science and Artificial Intelligence Laboratory</i>	
Designing High Performance Enterprise Wi-Fi Networks	73
<i>Rohan Murty, Harvard University; Jitendra Padhye, Ranveer Chandra, Alec Wolman, and Brian Zill, Microsoft Research</i>	
FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput	89
<i>Srikanth Kandula, Massachusetts Institute of Technology; Kate Ching-Ju Lin, National Taiwan University and Massachusetts Institute of Technology; Tural Badirkhanli and Dina Katabi, Massachusetts Institute of Technology</i>	
Efficiency Through Eavesdropping: Link-layer Packet Caching	105
<i>Mikhail Afanasyev, University of California, San Diego; David G. Andersen, Carnegie Mellon University; Alex C. Snoeren, University of California, San Diego</i>	

Large-scale Systems

Beyond Pilots: Keeping Rural Wireless Networks Alive	119
<i>Sonesh Surana, Rabin Patra, and Sergiu Nedeveschi, University of California, Berkeley; Manuel Ramos, University of the Philippines; Lakshminarayanan Subramanian, New York University; Yahel Ben-David, AirJaldi, Dharamsala, India; Eric Brewer, University of California, Berkeley, and Intel Research, Berkeley</i>	
UsenetDHT: A Low-Overhead Design for Usenet	133
<i>Emil Sit, Robert Morris, and M. Frans Kaashoek, MIT CSAIL</i>	
San Fermín: Aggregating Large Data Sets Using a Binomial Swap Forest	147
<i>Justin Cappos and John H. Hartman, University of Arizona</i>	

Thursday, April 17

Fault Tolerance

Remus: High Availability via Asynchronous Virtual Machine Replication161
Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, and Norm Hutchinson, University of British Columbia; Andrew Warfield, University of British Columbia and Citrix Systems, Inc.

Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures175
Chi Ho and Robbert van Renesse, Cornell University; Mark Bickford, ATC-NY; Danny Dolev, Hebrew University of Jerusalem

BFT Protocols Under Fire189
Atul Singh, Max Planck Institute for Software Systems and Rice University; Tathagata Das, IIT Kharagpur; Petros Maniatis, Intel Research Berkeley; Peter Druschel, Max Planck Institute for Software Systems; Timothy Roscoe, ETH Zürich

Monitoring and Measurement

Uncovering Performance Differences Among Backbone ISPs with Netdiff205
Ratul Mahajan and Ming Zhang, Microsoft Research; Lindsey Poole and Vivek Pai, Princeton University

Effective Diagnosis of Routing Disruptions from End Systems219
Ying Zhang and Z. Morley Mao, University of Michigan; Ming Zhang, Microsoft Research

cSAMP: A System for Network-Wide Flow Monitoring233
Vyas Sekar, Carnegie Mellon University; Michael K. Reiter, University of North Carolina, Chapel Hill; Walter Willinger, AT&T Labs—Research; Hui Zhang, Carnegie Mellon University; Ramana Rao Kompella, Purdue University; David G. Andersen, Carnegie Mellon University

Studying Black Holes in the Internet with Hubble247
Ethan Katz-Bassett, Harsha V. Madhyastha, John P. John, and Arvind Krishnamurthy, University of Washington; David Wetherall, University of Washington and Intel Research; Thomas Anderson, University of Washington

Performance

Maelstrom: Transparent Error Correction for Lambda Networks263
Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, and Einar Vollset, Cornell University

Swift: A Fast Dynamic Packet Filter279
Zhenyu Wu, Mengjun Xie, and Haining Wang, The College of William and Mary

Security

Securing Distributed Systems with Information Flow Control293
Nikolai Zeldovich, Silas Boyd-Wickizer, and David Mazières, Stanford University

Wedge: Splitting Applications into Reduced-Privilege Compartments309
Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp, University College London

Energy

Reducing Network Energy Consumption via Sleeping and Rate-Adaptation323
Sergiu Nedeveschi and Lucian Popa, University of California, Berkeley, and Intel Research, Berkeley; Gianluca Iannaccone and Sylvia Ratnasamy, Intel Research, Berkeley; David Wetherall, University of Washington and Intel Research, Seattle

Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services337
Gong Chen, University of California, Los Angeles; Wenbo He, University of Illinois at Urbana-Champaign, Jie Liu and Suman Nath, Microsoft Research; Leonidas Rigas, Microsoft; Lin Xiao and Feng Zhao, Microsoft Research

Friday, April 18

Routing

Consensus Routing: The Internet as a Distributed System351
John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, and Thomas Anderson, University of Washington; Arun Venkataramani, University of Massachusetts Amherst

Passport: Secure and Adoptable Source Authentication365
Xin Liu, Ang Li, and Xiaowei Yang, University of California, Irvine; David Wetherall, Intel Research Seattle and University of Washington

Context-based Routing: Technique, Applications, and Experience379
Saumitra Das, Purdue University; Yunnan Wu and Ranveer Chandra, Microsoft Research, Redmond; Y. Charlie Hu, Purdue University

Understanding Systems

NetComplex: A Complexity Metric for Networked System Designs393
Byung-Gon Chun, ICSI; Sylvia Ratnasamy, Intel Research Berkeley; Eddie Kohler, University of California, Los Angeles

DieCast: Testing Distributed Systems with an Accurate Scale Model407
Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat, University of California, San Diego

D³S: Debugging Deployed Distributed Systems423
Xuezheng Liu and Zhenyu Guo, Microsoft Research Asia; Xi Wang, Tsinghua University; Feibo Chen, Fudan University; Xiaochen Lian, Shanghai Jiaotong University; Jian Tang and Ming Wu, Microsoft Research Asia; M. Frans Kaashoek, MIT CSAIL; Zheng Zhang, Microsoft Research Asia

Index of Authors

Afanasyev, Mikhail	105	Kaashoek, M. Frans	133, 423	Reis, Charles	31
Andersen, David G.	105, 233	Kandula, Srikanth	89	Reiter, Michael K.	233
Anderson, Thomas	1, 45, 247, 351	Karp, Brad	309	Rigas, Leonidas	337
Badirkhanli, Tural	89	Katabi, Dina	89	Roscoe, Timothy	189
Balakrishnan, Hari	59	Katz-Bassett, Ethan	247, 351	Sekar, Vyas	233
Balakrishnan, Mahesh	263	Kohler, Eddie	393	Singh, Atul	189
Ben-David, Yahel	119	Kohno, Tadayoshi	31	Sit, Emil	133
Bickford, Mark	175	Kompella, Ramana Rao	233	Snoeren, Alex C.	105
Birman, Ken	263	Krishnamurthy, Arvind	1, 45, 247, 351	Subramanian, Lakshminarayanan	119
Bittau, Andrea	309	Lefebvre, Geoffrey	161	Surana, Sonesh	119
Boyd-Wickizer, Silas	293	Li, Ang	365	Tang, Jian	423
Brewer, Eric	119	Lian, Xiaochen	423	Vahdat, Amin	407
Cappos, Justin	147	Lin, Kate Ching-Ju	89	van Renesse, Robbert	175
Chandra, Ranveer	73, 379	Liu, Jie	337	Venkataramani, Arun	351
Chen, Feibo	423	Liu, Xin	365	Vishwanath, Kashi V.	407
Chen, Gong	337	Liu, Xuezheng	423	Vollset, Einar	263
Chun, Byung-Gon	393	Madhyastha, Harsha V.	247	Vutukuru, Mythili	59
Cully, Brendan	161	Mahajan, Ratul	205	Wang, Haining	279
Das, Saumitra	379	Maniatis, Petros	189	Wang, Xi	423
Das, Tathagata	189	Mao, Z. Morley	219	Warfield, Andrew	161
Dixon, Colin	45	Marchenko, Petr	309	Weatherspoon, Hakim	263
Dolev, Danny	175	Marian, Tudor	263	Weaver, Nicholas C.	31
Druschel, Peter	15, 189	Mazières, David	293	Wetherall, David	247, 323, 365
Feeley, Mike	161	Meyer, Dutch	161	Willinger, Walter	233
Gribble, Steven D.	31	Mislove, Alan	15	Wolman, Alec	73
Gummadi, Krishna P.	15	Morris, Robert	133	Wu, Ming	423
Guo, Zhenyu	423	Murty, Rohan	73	Wu, Yunnan	379
Gupta, Diwaker	407	Nath, Suman	337	Wu, Zhenyu	279
Handley, Mark	309	Nedevschi, Sergiu	119, 323	Xiao, Lin	337
Hartman, John H.	147	Padhye, Jitendra	73	Xie, Mengjun	279
He, Wenbo	337	Pai, Vivek	205	Yang, Xiaowei	365
Ho, Chi	175	Patra, Rabin	119	Zeldovich, Nickolai	293
Hu, Y. Charlie	379	Piatek, Michael	1	Zhang, Hui	233
Hutchinson, Norm	161	Poole, Lindsey	205	Zhang, Ming	205, 219
Iannaccone, Gianluca	323	Popa, Lucian	323	Zhang, Ying	219
Isdal, Tomas	1	Post, Ansley	15	Zhang, Zheng	423
Jamieson, Kyle	59	Ramos, Manuel	119	Zhao, Feng	337
John, John P.	247, 351	Ratnasamy, Sylvia	323, 393	Zill, Brian	73

Message from the Program Co-Chairs

NSDI '08 carries on the conference's tradition of presenting excellent and innovative work in the area of networked systems. We continue to take a broad view of that charter, selecting papers from across the range of the USENIX, SIGCOMM, and SIGOPS communities, rather than their intersection. The result is a strong program with a broad set of papers addressing topics from resource-rich corporate environments to challenging developing-region deployments, from application security to packet-filtering performance, and from monitoring, understanding, and debugging what is there to designing what is next.

We received 175 paper submissions, up from 110–120 in recent years. This big jump testifies to the success of the past four NSDI conferences in establishing this top venue for the networked systems community, but it left us scrambling a bit to cope with the increased volume while maintaining the standards that have made this conference such a success. We put out a call to past program committee members for assistance, and nearly everyone we contacted quickly agreed to help by serving as “Light” program committee members and providing thoughtful reviews for at least three papers.

All submissions were reviewed by several members of the program committee and selected external reviewers. Of the 175 submissions, about 100 moved on to receive a second round of reviewing. Altogether, the committee and reviewers completed 796 reviews to lay the groundwork for a ten-hour meeting in Austin, Texas, on December 17, at which the program committee selected 30 papers for the final program. Each of these papers was then shepherded by a program committee member.

We are grateful to everyone whose hard work makes this conference possible. Most of all, we are grateful to all of the authors who submitted their work to this conference. We thank the program committee for their dedication and hard work in reviewing an unexpectedly high number of submissions. We thank the PC-Light and external reviewers for lending their expertise on short notice. Thanks to the the USENIX staff for handling the conference logistics, marketing, and proceedings publication; it is a pleasure to work with such pros. We extend special thanks to Eddie Kohler for providing and supporting his terrific HotCRP reviewing system, and we thank Derek Murray for setting up and running it for us. We thank Derek Murray and Navendu Jain for acting as scribes at the PC meeting. Finally, we thank the NSDI '08 attendees and future readers of these papers: in the end, it is your interest in this work that makes all of these efforts worthwhile.

We look forward to seeing you in San Francisco!

Jon Crowcroft, University of Cambridge
Mike Dahlin, The University of Texas at Austin
NSDI '08 Program Co-Chairs

One hop Reputations for Peer to Peer File Sharing Workloads

Michael Piatek Tomas Isdal Arvind Krishnamurthy Thomas Anderson
University of Washington

Abstract

An emerging paradigm in peer-to-peer (P2P) networks is to explicitly consider incentives as part of the protocol design in order to promote good (or discourage bad) behavior. However, effective incentives are hampered by the challenges of a P2P environment, e.g. transient users and no central authority. In this paper, we quantify these challenges, reporting the results of a month-long measurement of millions of users of the BitTorrent file sharing system. Surprisingly, given BitTorrent's popularity, we identify widespread performance and availability problems. These measurements motivate the design and implementation of a new, one hop reputation protocol for P2P networks. Unlike digital currency systems, where contribution information is globally visible, or tit-for-tat, where no propagation occurs, one hop reputations limit propagation to at most one intermediary. Through trace-driven analysis and measurements of a deployment on PlanetLab, we find that limited propagation improves performance and incentives relative to BitTorrent.

1 Introduction

Peer-to-peer (P2P) networks have the potential to address long-standing challenges in networked systems. End hosts represent an immense pool of under-utilized bandwidth, storage, and computational resources that, when aggregated by a P2P network, can be used to absorb flash crowds, replicate data intelligently within the network, and externalize bandwidth costs. And unlike network-layer support such as IP multicast, P2P solutions can be deployed without architectural changes to the underlying network.

While significant progress has been made towards addressing the *technical* challenges of building P2P systems, their robustness ultimately depends on convincing users to contribute their resources, a challenge of *incentive design*. Early P2P systems such as Gnutella ignored incentives and were plagued by rampant free-riding, i.e., users consuming resources without contributing them [1]. Free-riding degrades system performance and limits scale. Subsequent systems such as BitTorrent explicitly built user contribution incentives into their design [4], but recent work has exposed methods of circumventing BitTorrent's incentives [11, 12].

Designing robust incentives for P2P networks is challenging due to the constraints of the environment:

- *No central control or trust*: Many practical problems

in P2P data sharing become trivial if we can assume a “*deus ex machina*”—some authority that can mint currency, perform accounting, and penalize miscreants. To date, P2P designs that rely on centralization of these tasks have not been widely adopted.

- *Open implementation*: Users are free to adopt any client implementation, even one that attempts to subvert incentives or strategize. This makes the P2P design challenge harder, as problems like free-riding can be defined away if all users must connect using a particular software release.

This paper concerns how best to design future incentive strategies for P2P networks. We proceed in two steps. First, to ground our work, we conducted a measurement study of BitTorrent. BitTorrent is a widely used P2P system and we were able to study the sharing behavior of tens of thousands of data objects and millions of users for more than one month. Surprisingly given BitTorrent's popularity, we identify widespread performance and availability problems, along with data on why these problems arise in practice. We find that problems cannot be wholly attributed to scarcity of potential data sources and/or capacity limitations. Instead, we argue that ineffective incentives account for the lack of resources, a point underscored by our measurement result that an average user joining an average swarm can get comparable download performance with only 1/100th the contribution.

A key reason for the weakness of current incentives is the duration for which they are active. Current incentives in BitTorrent operate within the context of a single object and only while clients are actively downloading. As a result, users have no reason to contribute once they have satisfied their immediate demands. This weakness implies the need for persistent incentives that operate across data objects and across time. Unfortunately, our measurements also show that most pairs of peers interact with one another in just one swarm, suggesting that long-term incentives will not arise from strategies based on direct interactions and local history alone. But, while most P2P users are transient, our study shows that a small minority of peers participate persistently and across many swarms; these users provide a scaffold for a solution.

The second part of the paper concerns our design of a solution to the problems we found in BitTorrent. We propose a new, *one hop* reputation protocol for P2P networks. Unlike digital currency systems, where contribu-

tion information propagates globally, or tit-for-tat, where no propagation occurs, one hop reputations limit propagation to at most one level of indirection. Surprisingly, this limited propagation suffices to provide wide coverage; we find that the majority of peers, while transient, have shared relationships through popular intermediaries one hop removed. We define a protocol that discovers these relationships, enabling a broad range of servicing policies using information beyond direct observations and local history. Through trace-driven analysis and measurements of a deployment on PlanetLab, we show that our default one hop system both improves performance for users in individual swarms and fosters the long-term incentives that are necessary for P2P systems to work well in the long run.

2 Sharing in the wild

To understand the real challenges facing P2P designers, we collected large-scale measurements of BitTorrent in the wild. Over the course of the study we observed more than 14 million peers and 60,000 swarms accounting for thousands of terabytes of transferred data. To measure the strength of contribution incentives, we joined real swarms, exchanged data at varying rates with peers, and collected information to distinguish unique users such as client software and version and IP address. We also tracked the popularity of swarms over time, recording both direct observations of peers and second hand accounts from coordinating tracker servers, membership DHT entries, and peer gossip messages.

Our measurements provide insights into the sharing workload that extend beyond the granularity of performance for a single user or behavior in a single swarm. Specifically, we show the following:

- Performance and availability in BitTorrent is extremely poor. The median download rate in observed swarms is 14 KBps for a peer contributing 100 KBps, and as many as 25% of swarms are unavailable.
- These performance and availability problems are not fundamental. Our measurements show that sufficient capacity is available to provide much better performance than is observed today, and many unpopular objects would see their availability improve if previous downloaders could be offered sufficient incentives to persist as replicas.
- Existing incentives in BitTorrent, while designed to encourage contribution, are largely ineffective. Because of the structure of the workload, BitTorrent incentives permit free-riding and strategic manipulation for the majority of BitTorrent swarms.
- Simple extensions to BitTorrent's incentive strategy, e.g., using direct long-term reciprocation for contributions, will not address the observed problems due to

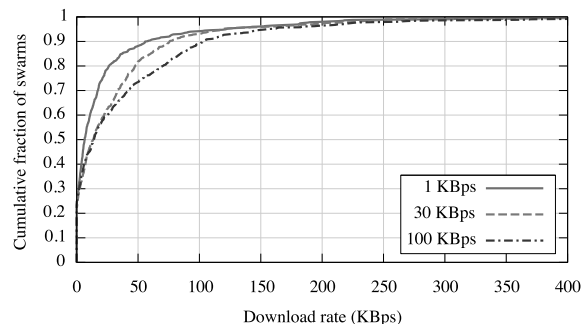


Figure 1: Download performance for different levels of contribution in BitTorrent. Each line gives the distribution of download performance for a contribution level as measured across thousands of real-world swarms in trace BT-1. Significant increases in contribution result in slight, if any, improvement in download performance.

a lack of repeat interactions in the sharing workload. However, the significant disparity in the popularity of P2P users points to the promise of new approaches based on indirect reciprocation.

2.1 Trace methodology

In BitTorrent, each file is split into blocks. Clients actively downloading a file are randomly matched, with matched peers exchanging data and control information as to which blocks they have and which they need. Ideally, a data source, or *seed*, only needs to provide each data block to a few random clients, and the rest of the work is done by the swarm of peers. Crucially, peers distinguish among competing requests for service according to a tit-for-tat policy: each client preferentially uploads blocks only to those peers that are actively providing data to it (and then, only to those that are providing data at the highest rate). Tit-for-tat is intended to provide better performance for peers that contribute more data.

The reported effectiveness of tit-for-tat has varied widely in existing work. Theoretical analysis, simulation and small testbed studies have pointed to its robustness [2, 10, 15] while more recent studies of performance in the wild have exposed circumstances under which tit-for-tat breaks down [11, 12, 16]. For system builders to design truly robust incentive protocols, a more complete understanding of P2P workloads is required. We collect and analyze BitTorrent trace data with the overarching goal of understanding when tit-for-tat incentives work and when they don't, in the wild.

We make reference to two traces of live BitTorrent swarms collected from a cluster of machines at the University of Washington. Between January 26th and February 3rd, 2007, we measured membership and download performance for instrumented clients participating in 13,353 swarms. We refer to this trace as BT-1. We collected a second trace, BT-2, from the same cluster

over the month of August 2007, providing measurements of 55,523 swarms. In both traces, every hour, a measurement coordinator crawled popular BitTorrent websites that aggregate information about new swarms, downloading all of these. Our instrumented clients joined these swarms periodically during the trace. We include information for only those swarms we successfully connected to at least once. To determine peer download rates, we measured the rate at which new blocks appeared in the peer's list of available blocks and also recorded availability of blocks. Each client contributed resources to the swarms at a rate of either 1, 30, or 100 KBps to examine the performance impact of varying the contribution level.

2.2 BitTorrent performance and availability

The download rate achieved by our measurement clients as a function of contribution rate is summarized in Figure 1 for trace BT-1. Even on the well-connected academic network used for our data collection, clients download slowly; contributing 100 KBps yields a median download rate of just 14 KBps, far short of saturating even a modest home broadband connection. Further, 25% of the time swarms were completely unavailable, i.e., delivered no data.

The poor performance of P2P networks cannot be explained by users simply lacking the capacity to offer peers a high average download rate, nor can poor availability be attributed to a long tail of fundamentally unpopular objects. Regarding performance, measurements of more than 100,000 BitTorrent peers in 2006 put average upload capacity at more than 400 KBps [8]. Skew in the capacity distribution is significant; the average value is roughly 10X the median. Regarding availability, our measurement results show that for many seemingly unpopular objects, the *existence* of replicas is not as much a problem as the *persistence* of replicas. The vast majority of swarms would have significantly more replicas if downloaders would simply continue to share after completing. We evaluate this by comparing available replicas assuming peers persist for either one day or one week after their initial observation. For trace BT-2, the median increase in available replicas is a factor of 3.

These measurements point to a problem of incentives. If users could be *convinced* to contribute all of their capacity, download performance would increase. If users were *convinced* to persist as object replicas, availability would improve. Realizing these benefits requires understanding the causes of today's weak incentives, the topic we turn to next.

2.3 Workload causes for weak incentives

The strength of a contribution incentive is the return it provides for contribution, i.e., the ratio of uploaded to

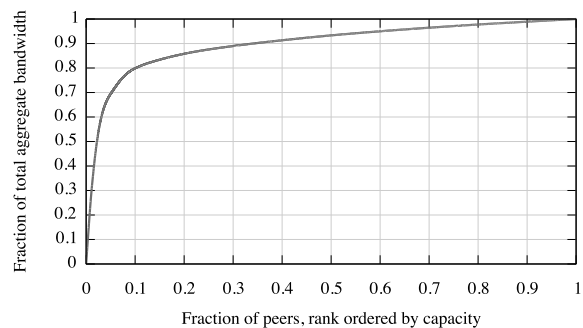


Figure 2: Cumulative fraction of total capacity (y-axis) attributed to the percentage of total peers rank ordered by capacity (x-axis). 80% of the total aggregate capacity of BitTorrent peers comes from the top 10% of users.

downloaded bytes. This section details two workload properties that weaken contribution incentives in BitTorrent. First, we examine how the distribution of bandwidth capacity among peers influences the incentive they have to make that capacity available. Second, we quantify the number of swarms for which random, altruistic contributions dominate performance.

Capacity: In BitTorrent, returns are known to diminish as contribution increases [12]. Peers at the low end of the capacity spectrum see large returns on their contributions, i.e., 10 bytes contributed might earn 15 reciprocated. This is balanced by reduced returns for peers with greater capacity. If the disparity between returns for high and low capacity peers were limited, contribution incentives would be only slightly weakened. In BitTorrent, however, the disparity is extreme. In our traces, increasing contributions 100-fold yields a 2-fold median marginal improvement in performance (shown for BT-1 in Figure 1).

The diminishing returns for contributions is particularly damaging for aggregate P2P resources as the majority of capacity is held by a small minority of users. Figure 2 shows the cumulative fraction of total capacity attributable to peers when ordered by individual capacity. If they were to contribute fully, the top 10% of peers would account for 80% of total capacity. Thus, for the highest capacity peers—those whose increased contribution would most help performance—the contribution incentive is weakest.

Altruism: A user's downloaded bytes come from either other peers actively downloading the object or seeds that have completed their downloads and continue to make data available. Because seeds do not have requests, they have no tit-for-tat basis for making servicing decisions, often doing so randomly in current implementations. An overabundance of seeds weakens contribution incentives as most users receive data regardless of contribution. Conversely, too few seeds also weakens incentives since

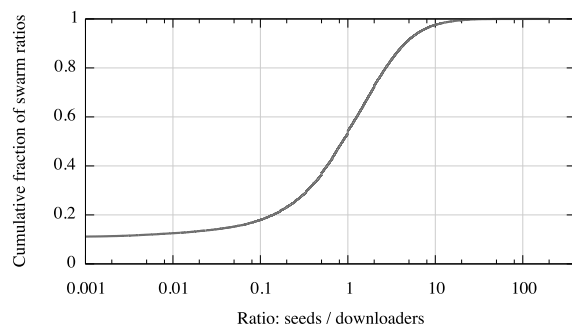


Figure 3: Ratios of seeds to downloading peers for swarms in BT-2. 11% of swarm observations showed no active seeds.

peers quickly run out of data to trade, becoming blocked.

Figure 3 summarizes the amount of seed-based altruism in our BT-2 trace. For each swarm, we compute the ratio of observed seeds and downloaders. This estimates the fraction of data a downloading peer is likely to receive at random, i.e., independent of contribution. The data shows that no one circumstance dominates. 11% of swarm observations show no active seeds (ratio 0) while 50% of swarms have just as many randomly contributing seeds as actively downloading peers.

Given the range of operating conditions we observe in practice, it is unsurprising that the BitTorrent performance picture is unclear. Some swarms enjoy a glut of altruistic donations, weakening contribution incentives and enabling free-riding. Other swarms are starved for data, causing performance to be constrained by availability rather than contribution. For the minority remaining swarms, the strength of the contribution incentive is tied to the bandwidth capacity distribution, with the majority of capacity being held by peers with little reason to contribute, leading to slow download rates.

2.4 A straw-man solution

In contrast to the standard game-theoretic tit-for-tat strategy, BitTorrent’s variant is rate-based. Instead of trading with peers byte for byte, reciprocation for a BitTorrent peer is decided only relative to its competitors and is apportioned equally among successfully competing peers. For instance, if a client C with capacity 20 receives data from peers X , Y , and Z at rates 5, 7, and 10 and selects only two peers at a time for reciprocation, C will send data to Y and Z at rate 10 apiece. This approach favors utilization over fairness and stateless operation over stability. Peers simply give away bandwidth if they are poorly matched in terms of rates and maintain only a short-term local history about each peer with which to make servicing decisions, switching peers frequently as short-term status changes.

An alternative to basing tit-for-tat decisions on *rate* is instead basing them on total data *volume*. This is the ap-

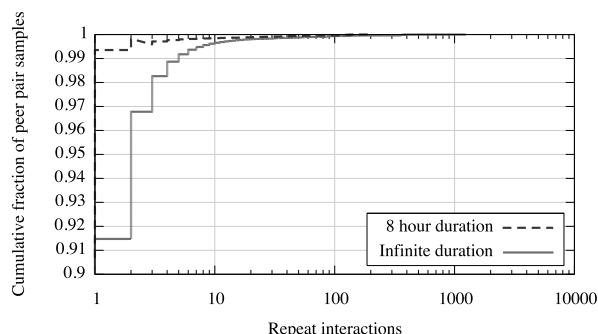


Figure 4: CDF of the frequency of repeat interaction between a pair of peers in the BT-2 trace. Note that the y axis is not zeroed and only pairs interacting at least once are considered. Even assuming infinite duration in swarms, 91.5% of interacting peers will do so only once. Limiting persistence reduces the chance of repeat interaction further to less than 1%.

proach taken by the eDonkey file sharing network, which stores per-peer state recording the amount of data sent and received, using this to rank peers with competing requests. From the perspective of strengthening contribution incentives, a switch from rate to volume seems promising, primarily because it offers the potential for long-term repeat interactions. Seeds might be willing to share files long after completion, improving availability, because in doing so they would contribute to peers whose memory of those contributions would induce reciprocation if the situation were reversed. Similarly, if high capacity peers were mismatched with low capacity peers, the contribution imbalance could be bounded or ignored—assuming repeat interactions would result in eventual repayment.

Unfortunately, volume-based tit-for-tat does not seem to have solved the performance problem in practice. Pucha et al. report a median download rate of 10 Kbps in the eDonkey network [14]—short of our observed median performance for BitTorrent swarms. Although numerous technical differences prohibit an apples-to-apples comparison, we hypothesize that the failure of volume-based tit-for-tat to promote contribution in eDonkey can be traced to a workload property that it likely shares with BitTorrent—a lack of pairwise repeat interactions.

We say that two peers share an interaction if either sends or receives data from the other. Peers exhibit repeat interactions if they exchange data in multiple swarms. Figure 4 reports the frequency of repeat interactions in the BT-2 trace, conditioned on a pair having interacted in at least one swarm. Because our trace data provides only coarse-grained observations of peer membership, i.e., we do not actively probe observed peers repeatedly to determine departure time, we give the distribution of repeat interactions assuming peers persist for either an

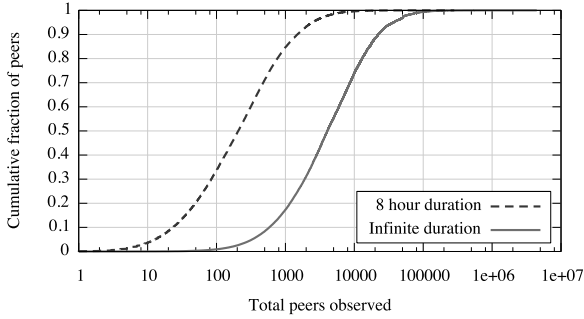


Figure 5: The distributions of peers encountered by BitTorrent users in the BT-2 trace. Whether assuming infinite or limited duration, a small minority of popular peers participates broadly.

infinite duration or an 8 hour interval. Assuming infinite duration overestimates the number of repeat interactions, while assuming an 8 hour duration may underestimate it for some long-lived peers. In either case, however, the chance of enabling long-term incentives via repeat interactions is slim. Even assuming infinite duration, more than 91.5% of peer pairs that occur in a single swarm do not arise in any other swarm over the course of our trace.

The apparent lack of repeat interactions suggests that direct, pairwise exchange based on local history alone will not suffice to enable the long-term contribution incentives needed to address the performance and availability problems we observe in the wild. But, although *direct* interactions appear insufficient, our workload measurements do provide a hint as to the effectiveness of *indirect* reciprocation; i.e., instead of peer *A* deciding whether to service the requests of *B* only on the basis of *B*'s contributions to *A*, indirect reciprocation might see *A* contributing to *B* due to *B*'s contributions to *C*, who has previously contributed to *A*.

Our data shows that most peers share an indirect relationship of this type. Further, a small number peers account for most of these intermediaries. 97% of all peers observed in trace BT-2 are connected either directly or through an intermediary among the most popular 2000 peers. This is due to a workload characteristic. Although most peers connect to only hundreds of other peers, a small minority is more extensively connected. Figure 5 shows the distribution of peer connectivity in trace BT-2.

The disparity in peer popularity is reflected in the distribution of demand as well. Figure 6 shows the distribution of total demand observed in our trace. We first ordered peers by popularity, i.e., the number of other peers with which they share a swarm. Next, we computed the cumulative fraction of demand attributable to these ordered peers. The top 25% of peers account for 78% of demand.

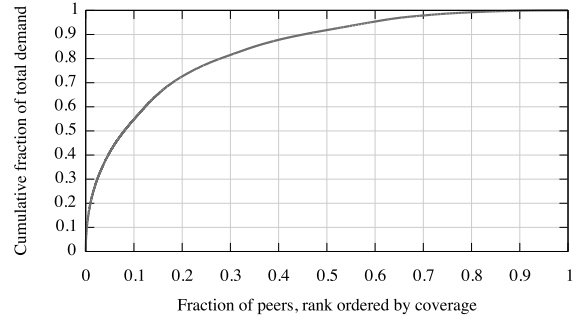


Figure 6: Cumulative fraction of consumption (y-axis) attributed to peers, ordered by popularity (x-axis).

3 One hop indirect reciprocation

Although repeat interactions are rare for the majority of peer pairs, a small minority of popular users have much wider coverage. In this section, we describe a new, *one hop* reputation propagation protocol designed to enable long-term reciprocation beyond this minority via indirect reciprocation. The main goal of one hop reputations is to foster persistent contribution incentives by recognizing and rewarding contributions made by users across swarms and over time. To achieve this, clients maintain a persistent history of interactions and, upon request, serve as intermediaries attesting to the behavior of others.

The key idea behind our scheme is to restrict the amount of indirection between contributing and reciprocating peers to at most one level of intermediaries. This restriction limits the propagation of information, promoting scalability, and allows for local reasoning about the trustworthiness of intermediaries, thereby fostering robustness.

While our measurements show that most peers share a one hop relationship, discovering and using these relationships requires more information than is available through direct observation alone. Peers need to name one another persistently across interactions and exchange messages about third party behavior. In Section 3.1, we define a protocol for exchanging the information required to discover intermediaries and to mediate indirect reciprocation.

Our protocol provides information but does not prescribe how that information must be used, separating the mechanism for exchanging information from the policy for using it. In Section 3.1, we specify a default policy designed to maximize coverage, i.e., the fraction of pairs of peers that can evaluate one another using one hop reputations. We also describe the resistance of our default policy to various forms of strategic manipulation, but we do not claim to be robust to all forms of attack. Instead, our design is intended to allow peers to freely evolve their strategies independently, and we consider several potential alternatives.

Notation	Definition
$n(x \rightarrow y)$	bytes sent directly from x to y
$n(x \leftarrow y)$	bytes received directly from y by x
$n(x \xrightarrow{y} *)$	bytes sent to other peers due to y 's recommendation as the intermediary
$n(x \xleftarrow{y} *)$	bytes received by x from other peers with y acting as the intermediary
$n(* \xrightarrow{x} y)$	summation of all bytes from any peer sent to y due to x 's referrals
$n(* \xleftarrow{x} y)$	summation of all bytes sent by y to each of x 's referrals
$\text{rate}(x \leftarrow y)$	the average rate at which y provided data to x

Table 1: State at client x for each peer y .

3.1 One hop reputation protocol

Our one hop reputation protocol can be broken down into two facets: the state maintained at each peer and messages used to propagate state between peers.

Per-peer state: One hop reputations extend volume-based tit-for-tat to incorporate reputation intermediaries. Intermediaries serve two purposes: bootstrapping connections between new peer pairs and maintaining accounting information regarding indirect reciprocation. Every client records each peer it has interacted with, either directly during data transfer or indirectly when that peer acts as an intermediary attesting to the behavior of others. Each peer is identified by a self-generated public/private key pair. While a peer can freely create new identities, our default policy rewards long-term persistence and includes provisions for mitigating Sybil attacks [6], creating little incentive to do so. Table 1 lists the state maintained by each client, which is indexed by the public key of its peers.

Figure 7 provides an example of the use of this information to bootstrap a new connection. In this case, a one hop intermediary I bootstraps the interaction between peers A and B who have not previously interacted. In the first two interactions, I exchanges data directly with A and B . These uninformed exchanges are infrequent and serve to bootstrap a reputation. At this point, I can serve as an intermediary between A and B . When they meet, A and B exchange control traffic (defined below) allowing them to recognize their common relationship with I . Because B has contributed to I in the past and A has received prior service from I , A can use its local history regarding I to inform its valuation of B .

Because the interactions between A , B , and C may not occur within the context of a single swarm, peers may need to contact intermediaries across multiple sessions. To aid in this, each client stores its current IP address and TCP port, indexed by its public key, in a DHT. Many popular P2P services already include a DHT, e.g.,

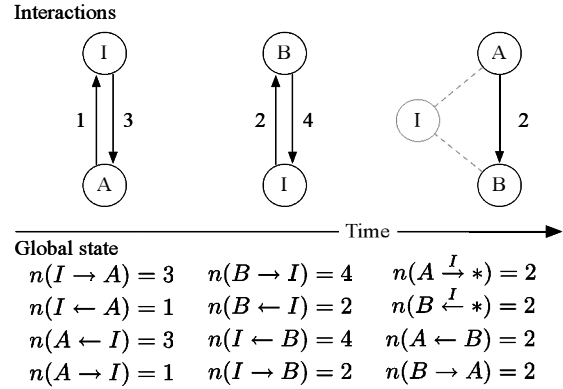


Figure 7: An example of peer state information used for bootstrapping. A recognizes B 's standing with intermediary I . Dashed lines indicate prior interactions.

Kademlia is used in BitTorrent and eDonkey. Although existing DHTs are generally robust, we do not evaluate their resistance to strategic or malicious behavior, instead opting to use provided values as hints only. Identity is independently verified using cryptographic keys and key \rightarrow IP mappings are locally cached.

State propagation: In the example of Figure 7, peer pairs A, I and B, I learn about one another directly through data transfer, requiring no explicit signaling. However, when A and B meet, they must exchange messages indicating which peers (possible intermediaries) they share in common and their status with those shared peers. In our protocol, this is a multi-step process.

1. First, peers order their local set of possible intermediaries to form what we refer to as their *top K set*. Inclusion in the top K set is a matter of local policy, but ordering this list by number of observations (our default policy) promotes the exchange of popular peers as intermediaries, increasing one hop coverage. Peers exchange topK messages upon connection.
2. Next, the intersection of local and remote top K sets is computed. This intersection is the set of shared peers that might be used as intermediaries for indirect reciprocation, but more information needs to be exchanged to compute the remote peer's reputation value.
3. For each possible intermediary, peers request attestation *receipts* of contribution by sending a receipt request message, containing the identity of the intermediary, to the remote peer. An attestation receipt for peer B from an intermediary I includes B 's local state at I , time stamped and signed by I 's private key.
4. Multiple peers can serve as intermediaries to mediate a specific interaction. If so, byte counts in local histories are updated fractionally based on the relative weight of each intermediary's valuation at the data source. This attribution message, a set of {identifier, weight} tuples, is sent to the receiving peer before data is transferred.

5. Once peers begin exchanging data, receipt messages are sent periodically from receiver to sender, to provide proof of received data and the corresponding increments to the sender's valuation. Before transmitting data, the sending peer dispatches a reserve message to mediating intermediaries containing the requesting peer's identifier and request size. These messages serve to preempt attacks based on using the recommendation of an intermediary multiple times and are optional. Periodically, update messages are sent to intermediaries, batching the reporting of transfers attributed to them, documented by received receipts.

In addition to facilitating identification of common intermediaries among peer pairs (Steps 1, 2), top K sets also bootstrap the local histories of new peers in the system. Each entry in the topK message contains a bit indicating whether the entry corresponds to an intermediary that can mediate a direct transfer or a gossip entry for a popular intermediary with whom the sender does not have a direct or indirect relationship. This is an optimization, effectively using two hop propagation to bootstrap one hop reputations. A client relying on direct, local observations alone to derive the coverage of potential intermediaries would have to directly observe them multiple times in distinct swarms before identifying those with high coverage. In the interim, new users would be unable to evaluate the quality of peers in good standing with popular intermediaries. Instead of relying on direct observations alone, peers may incorporate the gossip intermediaries included in the top K sets of their directly connected peers, combining this information with direct observations in their local history. Care must be taken when incorporating this information to prevent strategic manipulation, an issue we will discuss later.

3.2 Policies

Our state exchange protocol provides information about peers that enables a range of valuation policies. In this section, we propose a default policy designed to maximize coverage. However, peers are not required to follow this policy, and we present it as just one plausible design point among several alternatives. Other options are possible, e.g., trading coverage for resistance to strategic manipulation and collusion.

3.2.1 Default policy

Our default policy is the indirection-enabled analogue of volume-based tit-for-tat. When a peer makes servicing decisions, it restricts contribution to only those peers who have a positive or near positive "balance" with the system as a whole. This limits the potential for free-riding, if most peers have a one hop basis for making decisions. In this section, we define precisely how each client ranks the requests of others using one hop infor-

Default servicing policy of peer L

```

1: for each peer  $P$  requesting service
2:   if  $P$  and  $L$  have interacted directly
3:      $Reputation_P \leftarrow$  Computed using Equation 4
4:   else
5:      $topK_P \leftarrow$  Send topK request
6:      $mediators_P \leftarrow \text{random\_subset}(topK_P \cap topK_L)$ 
7:     for each intermediary  $I \in mediators_P$ 
8:       Send receipt request to  $P$  for  $I$ 
9:     done
10:     $Reputation_P \leftarrow$  Computed using Equation 3
11:   fi
12: done
13:  $N \leftarrow \sum_{P | Reputation_P > 1.0 - \epsilon} Reputation_P$ 
14: for each peer  $P \mid Reputation_P > 1.0 - \epsilon$ 
15:    $N_P \leftarrow \sum_{I \in mediators_P} w_L(I)$ 
16:   Send Attribution message,  $\forall I \in mediators_P, \{I, \frac{w_L(I)}{N_P}\}$ 
17:   Send data to  $P$  at rate  $\sim \frac{Reputation_P}{N}$ 
18: done

```

Figure 8: The default one hop servicing policy.

mation and how membership of possible intermediaries in the top K set is decided.

Computing reputations: The value of a one hop reputation for peer B from the perspective of a peer A is determined by three factors: 1) the volume of data exchanged between A and B (if any), 2) A 's valuation of B 's attesting intermediaries, and 3) B 's reputation with each attesting intermediary. We denote A 's valuation of intermediary I as $w_A(I)$ and the valuation of peer B at intermediary I by $v_I(B)$, defining these precisely in Equations 1 and 2, respectively.

$$w_A(I) = \frac{n(A \leftarrow I) + n(A \xrightarrow{I} *)}{n(A \rightarrow I) + n(A \xrightarrow{I} *)} \quad (1)$$

$$v_I(B) = \frac{n(* \xleftarrow{I} B) + n(I \leftarrow B)}{n(* \xrightarrow{I} B) + n(I \rightarrow B)} \quad (2)$$

These expressions allow us to define the indirect reputation value of a peer B from the perspective of a peer A , $ivalue_A(B)$, given a set of mutually recognized intermediaries, \mathbf{I} , as:

$$ivalue_A(B) = \frac{\sum_{I \in \mathbf{I}} w_A(I) \times v_I(B)}{|\mathbf{I}|} \quad (3)$$

If two peers have a bidirectional relationship, the direct reputation value, $dvalue_A(B)$, is defined as:

$$dvalue_A(B) = \frac{n(A \leftarrow B)}{n(A \rightarrow B)} \quad (4)$$

Figure 8 shows how servicing decisions are made regarding a set of peers requesting data. Our default policy uses direct observations if they exist, relying on one hop indirection only if local history is unavailable (lines 2–4). This gives peers an incentive to operate as an intermediary since doing so increases their value across all peers directly, removing the need to rely on one hop coverage and preempting other peers that need to compete

based on indirect evaluation. If indirection is required, we use a randomly chosen subset¹ of shared intermediaries to mediate transfers (line 6). Attribution receipts are requested for each mediator (lines 7–9) before computing indirect reputation. After reputations have been computed, requests can be serviced. We impose a reputation threshold to limit contribution imbalance (line 14). Selected peers receive Attribution messages indicating the fraction of throughput to account for each mediator (line 16), normalized by the weight of all mediators (line 15). Servicing rates are assigned proportionally based on relative reputation (line 17), normalized (line 13) across the set of selected peers.

Top K membership: Our default policy for populating top K sets is based on the number of direct and indirect observations of each potential intermediary. When a client directly observes a peer, its occurrence count is incremented by one. In addition to direct observation, our default policy also integrates indirect observations in the form of top K sets from peers. In this case, occurrence counts are updated fractionally, weighted by the number of received bytes from the peer reporting an observation relative to others in the recent past.

If an intermediary is unavailable or refuses an update message, its occurrence count is reduced by 20% or 2, whichever is larger. This AIMD policy is intended to promote agreement on intermediaries with wide coverage while quickly pruning popular peers that become overwhelmed or unavailable. Overhead concerns are treated further in Section 4.1.

Liquidity: Peers have an incentive to keep in good standing with intermediaries that have high coverage. Peers gain standing with a popular intermediary by either satisfying its direct requests (direct contribution) or contributing to peers that have satisfied the intermediary’s requests one hop removed (indirect contribution). In the former case, there is a net increase in the sum total of reputation values at the intermediary. In the latter case, the reputation of one peer is simply transferred to another. Thus, the sum total of reputation values at an intermediary—the *liquidity* the intermediary provides the system—is limited by the intermediary’s demand. This can result in a disabling shortage. Two peers may share many intermediaries that cannot be used because of a lack of standing with those intermediaries, reducing the *effective* coverage of otherwise popular intermediaries. This situation will arise unless popular intermediaries generate enough demand to allow one hop trading in satisfying their requests to cover the remainder of demand in the system.

To address this problem, the demand recorded in attestation receipts obtained for direct contributions is inflated

by a fixed amount by all intermediaries. Because the inflation factor depends on the fraction of total demand generated by popular intermediaries, its value is workload dependent. In our traces, the most popular 2000 peers account for 1.6% of total demand, suggesting that an inflation factor of 100 provides sufficient liquidity.

Although the demand of the minority of popular users relative to total user demand varies little over the course of our trace, this may not be a reliable workload characteristic. If fixed, a static inflation factor will suffice to maintain sufficient liquidity even as users join and leave the system. If not, intermediaries will need to adjust their value at the cost of introducing true economic inflation into the system. This requires only a policy change. Intermediaries can mint receipts with higher or lower byte values which peers can recognize and incorporate into their valuation of intermediaries.

Intermediary incentives: In addition to providing sufficient liquidity, inflating the value recorded in attesting receipts also creates an incentive to serve as an intermediary. In the common case that two peers have not directly interacted, their valuation of one another is based on standing with popular intermediaries, trading in indirect attestations 1:1, i.e., 1 byte contributed for 1 byte attested. But, satisfying an intermediary’s requests *directly* results in a 1: N exchange, where $N > 1$ is the inflation factor of intermediary receipts. Because of their higher returns, peers prioritize the requests of popular intermediaries. This preferential treatment requires an intermediary to continue mediating transactions: if it stops responding to queries and updates, it will be pruned from the set of preferred intermediaries by peers that it ignores.

3.2.2 Alternate policies

A large body of work on P2P reputation systems has documented a well-known set of challenges for incentive design. These include bootstrapping new users, Sybil attacks, collusion, and free-riding. To date, no comprehensive solution has emerged that addresses all of these issues, nor do we claim that our approach does. Instead, we have explicitly designed our system to separate the protocol mechanisms of reputation propagation and maintenance from the policy for acting on that information. As a result, our scheme supports a range of policies operating at different levels of vulnerability to well-known attacks. Our measurements of BitTorrent suggests that vulnerability to attack is a negative attribute, but not necessarily a fatal one. In this section, we detail several of these policies and the risks they carry.

Direct, deficit 1 block-based tit-for-tat: This is the most conservative policy we consider, ignoring most available information in the interest of (near) strategy-proof operation. Peers make the positive first step in the

¹Size 10 in our implementation, see Table 2.

traditional tit-for-tat game, sending at most one unreciprocated data block to a peer. For strategic adversaries, attacks are limited. Free-riders obtain at most one block, and while they might collect many blocks by repeating the game with a large number of peers, our measurements show that most swarms have hundreds of peers or fewer while most objects are comprised of thousands of data blocks. Sybil attacks are similarly frustrated. Collusion carries little benefit—the valuation of a peer is based only on the directly observed behavior of that peer. Bootstrapping is straightforward, as adherents to this strategy willingly contribute the single data block needed to start playing the game. Finally, unfairness in data exchanged is sharply bounded per-peer. The strategic robustness of this approach comes primarily at the cost of a lack of long-term incentives; seeds would limit their contribution to just one block, further reducing availability.

Direct, volume-based tit-for-tat: This strategy eliminates the bound on unfairness in block deficit tit-for-tat. Peers contribute their full capacity, realizing that free-riders / Sybil identities might never reciprocate and seed contributions may never be repaid due to the small chance of repeat interactions. Willingness to make such contributions increases utilization while retaining the collusion resistance of local reasoning as in deficit tit-for-tat. However, because repeat interactions are infrequent, long-term contribution incentives remain weak.

Indirect contribution, reputation $> 1.0 - \epsilon$: This is our default policy. The value of ϵ controls the level of indirect imbalance a peer is willing to tolerate. However, because one hop reputations do not provide precise global accounting, peers contributing data due to third-party standing accept the risk that the intermediaries they choose to mediate an exchange may not have wide coverage. But, as we will show in Section 4, most one hop interactions can be mediated by multiple intermediaries with wide coverage for observed workloads.

Indirect / random excess contribution: For many types of strategic behavior, e.g., free-riding, limiting damage depends on reducing contributions when the reputation of a peer cannot be reliably ascertained. Much like the utilization / robustness tradeoff of deficit n tit-for-tat, peers are faced with a choice when considering what to do with any excess capacity that remains after servicing all requests based on one hop information. Continuing to service requests—essentially at random—enables free-riding behavior, with its effectiveness growing as the amount of random contributions increases.

3.3 Attacks and defenses

Permitting indirection greatly expands the range of attacks available to a strategic client or a set of colluding clients. We consider several attacks, but do not claim to make indirect contribution under our default policy fully

resistant to all forms of strategic or malicious behavior. For increased robustness, clients are free to adopt an alternate policy that is more conservative.

- *Intermediary collusion to promote peers:* A popular intermediary may collude with peers or with Sybil identities by providing falsely generated attestation receipts. The effectiveness of this attack is ameliorated by the need for the intermediary to have contributed widely to become popular in the first place. In a sense, its good standing with others is its own reputation to lose, and if it does not continue to directly maintain its standing with enough users through continued contributions, the value of good standing with it will diminish, similarly diminishing the value of its falsified receipts.
- *Peer collusion to promote intermediaries:* Because peers prioritize the requests of popular intermediaries in order to gain receipts with high coverage, colluders may attempt to promote a manufactured identity that has not contributed widely. However, our one hop restriction requires directly verifiable contribution to carry out this deception. Because the integration of external top K sets with a peer's local history is weighted by the contributions of the reporting peer, members of the colluding set must “pay” an unknown amount for the promotion of their fraudulent intermediary, balancing the uncertain returns of the scheme against the initial contributions required to carry it out.
- *Peer collusion to defraud intermediaries:* A peer may collude with others or with Sybil identities to report false contributions to inflate standing with popular intermediaries. For example, a peer A that has legitimately contributed 1 MB to popular intermediary I could falsely report contributions of 1 GB to colluding identity B , generating 1 GB worth of indirect contribution through I for peer A . Our default intermediary policy simply disallows the case of negative net contribution enabling this attack, meaning that A can transfer the attribution of its 1 MB worth to B but cannot exceed the amount of data that I can directly verify A has contributed.

4 Evaluation

Comprehensive evaluation of incentive systems is often frustrated by an overabundance of metrics. For example, protocol designers can choose among fairness, utilization, bootstrapping time, overhead, and resistance to strategic or malicious behavior. As we observed in Section 3.3, optimizing for one of these metrics often comes at the expense of another. Further, evaluating the ability of an incentive strategy to promote long-term incentives—a necessity for increasing availability in P2P systems—requires a model of user behavior that is itself

Value	Definition
2000	The number of peers in a top K set
10	The maximum number of potential intermediaries in overlapping top K sets used to mediate transfers
10 MB	Data exchanged before intermediary synchronization updates
100	The multiplicative factor of reported bytes in intermediary receipts
0.1	ϵ bound on reciprocation

Table 2: One hop reputation parameters and values used in our evaluation.

long-term.

In this section, we provide a threefold analysis of our system to partially address these evaluation challenges. First, we describe the key parameters of our prototype implementation. These parameters control overhead, which we evaluate for our observed workload. Second, we use trace data to examine the coverage of one hop reputations; coverage distills many existing metrics for the quality of reputation systems including bootstrapping time, susceptibility to free-riding, and return on investment. Finally, we report experimental results obtained on PlanetLab using a prototype implementation of our system that we have layered on top of the popular Azureus BitTorrent client. Our PlanetLab experiments measure the real-world performance improvement that can be obtained by using the additional information one hop reputations provide.

4.1 Implementation parameters and overhead

Section 3 describes how reputation information is propagated and used by peers, but we have deliberately delayed assigning several workload-dependent parameters to provide context. These parameters and the values assigned in our prototype are listed in Table 2. We consider the influence of each of these parameters.

Top K set size: The exchange of top K sets serves two purposes. First, it allows peers to agree on shared intermediaries for data exchange. Second, it allows peers to quickly learn which intermediaries have wide coverage (and are therefore most valuable). When K is large, peers have a higher chance of discovering shared intermediaries and information about intermediaries is propagated more rapidly. In our implementation, peers exchange top K sets of size 2000. As each entry in the set is a 128 bit public key identifying an intermediary and a gossip bit, bidirectional exchange requires less than 64 kilobytes of data per peer connection, a small fraction of the megabytes of object data typically transferred between peer pairs.

Synchronization, mediating intermediaries: The intersection of top K sets forms a set of possible intermedi-

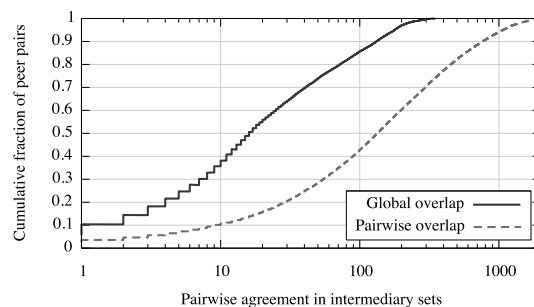


Figure 9: Sizes of overlapping top K sets. Pairwise overlap gives the distribution of overlap sizes for randomly chosen pairs of peers in the BT-2 trace. Of these intersection sets, global overlap shows the number shared with the top 2000 intermediaries overall.

aries that can facilitate indirect reciprocation. Of these, peers in our implementation select a random subset of size 10 to act as the mediators for the transfer. Clients synchronize state with intermediaries after every 10 MB transferred. This parameter controls the update burden on the most popular intermediaries, a potential scalability bottleneck.

We measure the number of updates at popular intermediaries using our trace data. Total demand of the 14 million peers in our trace, calculated by counting distinct peers in each swarm and multiplying by swarm file size, is 26,752 terabytes (roughly 2 GB per user). Assuming perfect agreement and static membership in top K sets, each of the most popular intermediaries will need to process 1.4 million updates $\left(\frac{26752 \text{ TB}}{2000 \times 10 \text{ MB}}\right)$. Updates are signed and include a hash (16 bytes), timestamp (4 bytes), 128 bit sender and receiver public keys (32 bytes), and bytes sent and received (16 bytes, 68 bytes total). These updates will be distributed over intermediaries, yielding an overhead of 3 MB per day for each popular intermediary $\left(\frac{1.4 \text{ million} \times 68 \text{ B}}{31 \text{ days}}\right)$. In practice, individual peers will differ in their views of the quality of intermediaries, reducing load, and will also differ in their relative share of total demand and hence update traffic. Also, data exchange between actively downloading peers will be mediated by direct tit-for-tat after the initial exchange, further reducing load. Finally, the size of top K sets can be increased if necessary to further distribute load.

4.2 One hop coverage

For our system to work well, the key factor is whether or not the majority of interactions have a one hop basis for computing reputations. In short, do one hop reputations provide good coverage? We find that they do, arriving at this conclusion using our BT-2 trace of peer interactions to examine the number of overlapping peers in randomly chosen top K sets, assuming all peers use one hop repu-

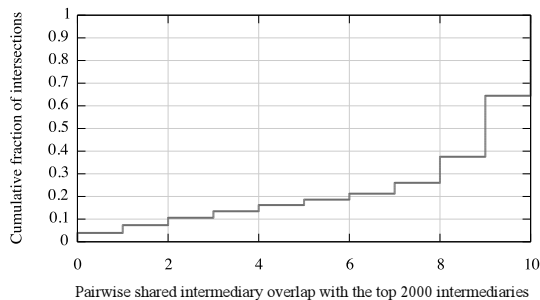


Figure 10: The distribution of intermediary overlap between the top 2000 intermediaries and 10 randomly chosen shared intermediaries between randomly chosen peers. Mediating transfer through a small subset of shared intermediaries suffices to provide wide coverage.

tations and the parameters of Table 2. Figure 9 shows the number of shared intermediaries between two randomly chosen peers with local histories built up according to our trace. This data indicates the amount of local history that peers can build. Some users participate in only a few swarms, while others participate in hundreds. Applying one hop reputations provides a measure of both coverage and convergence. Coverage is measured by pairwise overlap among the top K sets of randomly matched peers. The median number of shared intermediaries is 83 and more than 99% of peers have at least one common entry in their top K sets. The most useful shared intermediaries are those with wide coverage, and we say that a top K set has converged if it overlaps with the most widely used intermediaries measured over the top K sets of all peers. For some long-lived peers that participate in many swarms, convergence is high, but other peers participate in only a few swarms, limiting their view. When intermediaries with relatively limited coverage mediate transfers, the potential for returns is diminished. Fortunately, our data shows that most randomly matched peers share several intermediaries that are among the 2000 with widest coverage (Global overlap, Figure 9).

Most peers have several choices when deciding which intermediaries to use to mediate their transfers. Using all available intermediaries or only several of the most popular distributes the risk of choosing an intermediary with poor coverage, but contacting potentially hundreds of shared intermediaries per-peer increases overhead. Popular intermediaries that become overloaded may refuse updates from peers that generate too many. To avoid this, we evaluate a default policy of randomly choosing a maximum of ten intermediaries from the shared set for randomly paired peers. Section 4.1 describes how this limit controls overhead. In Figure 10, we examine whether good coverage can be maintained given this limit, finding that even when randomly subsampling shared intermediaries, most interactions will still

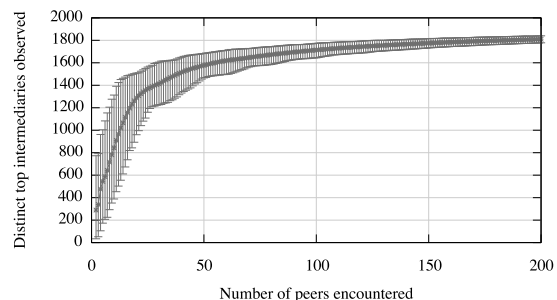


Figure 11: The number of top 2000 intermediaries observed by a new peer as a function of peers directly contacted, averaged over 100 trials with error bars showing the 5th and 95th percentiles.

be mediated through several with wide coverage. 96% of randomly chosen peer pairs share an intermediary that is among the 2000 most popular when randomly subsampling, with a median overlap of size 9.

In the remainder of this section, we describe the implications of high one hop coverage on three properties: bootstrapping time for new users, free-riding, and return on investment for contributions.

Bootstrapping new users: Coverage of popular intermediaries and convergence of top K sets controls the bootstrapping time of one hop reputations. The results of Figures 9 and 10 demonstrate that agreement among top K sets is high, assuming local history built up according to our trace. This includes peers that have participated in the system *at least once*. We next examine how quickly one hop reputations can bootstrap *new* peers that have no local history. Bootstrapping a one hop reputation is a two step process. First, clients need enough observations to ascertain which intermediaries have high coverage. Second, they need to encounter peers that have established relationships with high coverage intermediaries. We consider both aspects.

- *How quickly can a new peer determine intermediary value?* We answer this question statistically using trace data. First, a new identity with an initially empty top K set is created. Next, as in previous experiments, we use our trace data to build up representative top K sets for peers already in the system. We then sample these top K sets randomly, integrating them with that of the newly created identity using randomly assigned weights drawn from our measured end-host capacity distribution of BitTorrent peers. The results of this process are summarized in Figure 11, which gives the number of entries in a new user's top K set that are shared with the 2000 most popular intermediaries globally as a function of the number of peers observed. Data points are averaged over 100 trials with error bars showing the 5th and 95th percentiles. These results demonstrate the rapid bootstrapping of one hop rep-

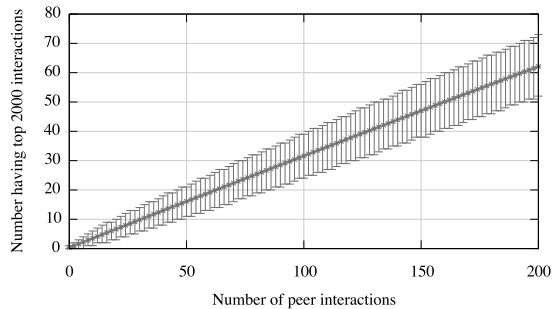


Figure 12: For a new user, the number of peers observed having a direct or one hop relationship with the top 2000 peers as a function of contacted peers.

utations under today’s workloads. After only a few dozen interactions with randomly chosen peers, a new user can identify intermediaries with wide coverage.

- *How quickly can peers gain standing with popular intermediaries?* Simply identifying the value of intermediaries does not suffice to enable one hop trading. New peers must encounter and exchange data with popular intermediaries directly or indirectly through others that have directly interacted with them. Figure 12 gives the number of the top 2000 intermediaries in our trace encountered either directly or indirectly by a new user as a function of the number of peers the new user encounters. This data is a conservative bound since we do not model the transfer of standing with the top intermediaries that would occur over time. As with Figure 11, we compute this data statistically, averaging over 100 trials. This data shows that peers observe popular intermediaries either directly or indirectly frequently, allowing a new peer to quickly trade via intermediaries with wide coverage.

Taken together, these results show that new users are likely to both encounter an opportunity to gain standing with a popular intermediary (Figure 12) and recognize that opportunity (Figure 11).

Free-riding: The coverage achieved by one hop reputations for today’s workloads suggests that free-riding can be deterred without the significant sacrifices in utilization required by schemes such as deficit 1 block-based tit-for-tat. Because peers usually have a one hop basis for making servicing decisions, the majority of each user’s capacity can be allocated to peers that can demonstrate their contributions. However, we do not claim that one hop reputations *prohibit* free-riding, as selfish peers in large swarms may still be able to scavenge enough altruistic excess capacity to complete file downloads. Rather, our goal is simply to limit the opportunities for effective free-riding by providing peers with more information. Under today’s reputation systems, the random contributions that enable free-riding are necessary because obtaining information about good peers requires making

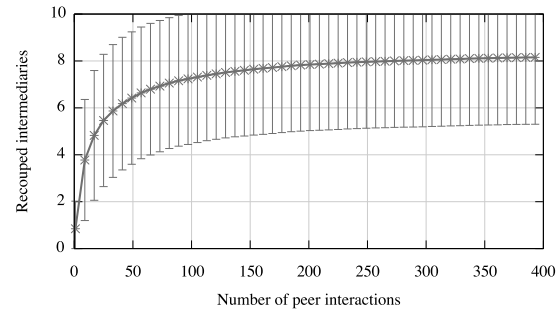


Figure 13: The number of mediating intermediaries from a randomly drawn one hop transfer encountered in subsequent interactions as a function of the number of subsequent interactions. A contributing peers sees higher returns on investment if mediators are chosen that will reappear quickly. Error bars give the standard deviation.

contributions to all peers, beneficial or not.

Return on investment: The return on investment for contributed bytes is the amount of reciprocated bytes generated by that contribution. For reputations that persist over short time periods, as in BitTorrent’s tit-for-tat, return on investment is immediate and can be measured or computed. For persistent reputation schemes, however, return on investment can be a misleading measure of incentive strength. For instance, volume-based tit-for-tat maintains state regarding each contribution, providing 1:1 returns on all contributions eventually if peers do not leave the system permanently and continue to make requests. But, as we observed in Section 2.4, repeat, direct interactions are extremely rare for today’s workloads, suggesting that peers would need to tolerate lengthy delays before receiving reciprocation. Particularly in P2P networks, waiting for reciprocation opportunities dampens returns as some peer departures are permanent.

Because one hop reputations have persistent memory, we evaluate the returns from contribution in terms of the number of interactions required to recoup contributed bytes. Each time a peer contributes data due to indirect, one hop standing, that contribution is mediated through a subset of the shared intermediaries between sender and receiver. The contributing peer earns reciprocation for that contribution only if it later can use some of those intermediaries to mediate another transfer where it acts as receiver. A contributing peer will see poor returns if it selects a mediating intermediary that has poor coverage.

We use the one hop reputations for peers in our BT-2 trace to compute the number of interactions required to repeat the use of an intermediary from the set mediating a random initial contribution. Figure 13 shows results averaged over 1000 trials with error bars showing standard deviation. For each sample, we compute a size 10 random subset of shared intermediaries between two randomly drawn peers, interpreting this set as the me-

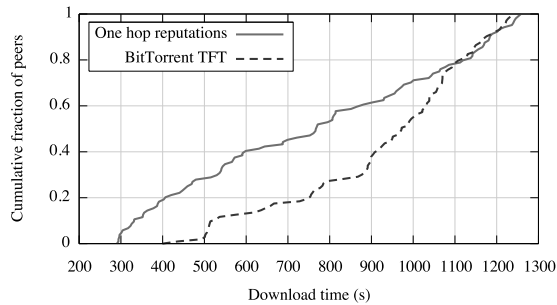


Figure 14: A comparison of performance for bulk file distribution on PlanetLab. Leveraging the historical rate information provided by one hop reputations improves performance.

diating intermediaries in a potential transfer. We then repeat this process, computing the overlap between subsequent mediating sets and the original contribution set. Figure 13 shows that, on average, peers will encounter 8 of the 10 initially used intermediaries within a few hundred peer interactions. Although this implies that returns for one hop contributions are not 1:1 for our default policy, peers do see a higher opportunity for returns on contributions when compared with direct, long-term reciprocity schemes that require tens of thousands of interactions before payback occurs, if at all.

4.3 Deployment on PlanetLab

Our evaluation thus far has focused on the ability of one hop reputations to promote strong contribution incentives through wide coverage and returns on contribution, improving performance by providing users with a reason to contribute more capacity. In this section, we focus on the concrete performance improvement one hop reputations can provide, regardless of strengthened incentives.

Because one hop reputations include not only an accounting of transfers but also the *rate* of those transfers (ref. Table 1), users can make intelligent decisions about which peers are likely to disseminate data rapidly. In BitTorrent today, peers do not maintain historical information about peer capacities and must rely on tit-for-tat to funnel data to high capacity peers. Unfortunately, high capacity goes unnoticed by tit-for-tat when the data available to trade is the limiting factor for performance, as is the case when a file first becomes available or when seed capacity is limited. In both cases, quickly utilizing the full capacity of the swarm depends on high capacity peers receiving data first so they can quickly replicate it, reducing the amount of time required for other peers to gain data to trade.

To measure the performance benefit realized by using rate information, we deployed our prototype one hop reputation implementation on PlanetLab, comparing its performance with the original Azureus BitTorrent imple-

mentation on which it is layered. Figure 14 compares the completion times for 100 peers downloading a 25 MB file using BitTorrent in one trial and one hop reputations in the next with simultaneous arrivals in both trials. Before conducting the one hop download trial, we first primed the local histories of participants by distributing a different 25 MB file. We record the download times required to download the second file while using the local history built up during the priming run. To adhere to the skewed bandwidth distribution typical of end-hosts in BitTorrent swarms, we used application level bandwidth capacity limits with values drawn from the percentiles of the end-host capacity distribution for BitTorrent clients given in [12].

One hop reputations improve performance for roughly 75% of PlanetLab hosts, providing a median reduction in download time from 972 seconds to 766 seconds. This performance improvement is attributable to the ability of historical information to allow peers to quickly find good tit-for-tat peerings. This is particularly true for the seed, which distributes data randomly in the reference implementation of BitTorrent. Rather than relying on random selection, a one hop seed can preferentially give data to users it knows have high capacity. These peers amplify the initial contributions of the seed, pumping data into the systems rapidly and increasing utilization relative to that of random selection, which may give initial data to slow peers that cannot quickly replicate it.

5 Related work

Our focus on incentives has led us to build a protocol layer for exchanging peer reputation information, one that can be shared across time and across content distribution applications. While incentives could be added to any content distribution system, it can be quite difficult to design a robust incentive system when participation is ephemeral and identities are not persistent, as we have seen in BitTorrent.

The research community has made considerable progress towards understanding BitTorrent dynamics; we use many of these insights in the design of our one hop reputation system. Qui and Srikant [15] analytically model the BitTorrent protocol, showing that in certain conditions, it achieves a Nash equilibrium. Unfortunately, our measurements show that these conditions are typically not met in practice in live BitTorrent usage. Bharambe et al. [2] use simulation to show that BitTorrent engages in progressive taxation, taking from high capacity peers to give unreciprocated bandwidth to low capacity peers. BitTyrant [12] exploits this observation, showing that clients can strategically deploy their upload bandwidth to significantly improve their local performance, and in the bargain, reduce performance of the swarm. Locher et al. [11] and Sirivianos et al. [16]

made similar points, showing that BitTorrent provides weak protection against free riding clients. Our measurement results of BitTorrent in the wild are compatible with the results of those papers, expanding on previous studies of only a subset of swarms with a large number of active downloaders. Our data is broader, showing that in most BitTorrent swarms incentives are inoperable. Wang et al. [18] and Tribler [13] argue for using third party helpers to improve client performance in BitTorrent. While we show that increased upload contribution only marginally improves download rates in BitTorrent, instead we generalize the notion of helpers, using one hop reputations to provide an incentive for third parties to do work on behalf of others.

Our work has the most in common with recent work on the design of reputation systems for various P2P applications. Karma [17] focuses on building a robust, incentive compatible distributed hash table as a basis for trading a digital currency. DHTs are a particularly difficult venue for robust incentives, as peers are both ephemeral and have little repeated interaction. To address this, Karma sets up a replicated system of banks on top of the DHT to serve as reputation authorities. In our one hop reputation system, popular nodes serve as a kind of ad-hoc bank without any additional mechanism beyond peer gossip of popular nodes and signed receipts. Our use of indirection is similar in some respects to EigenTrust [9] and multi-level tit-for-tat [21]. EigenTrust focused on the problem of inauthentic files, computing a global reputation for every participant. Reputations in our system are local, and clients are free to evolve their strategy independently over time. Multi-level tit-for-tat demonstrated that much of the value of EigenTrust can be achieved with only a few levels of indirection, an insight we use in the design of one hop reputations.

Finally, we observe that our protocol for propagating reputations allows peers to make their own policy decisions, making it possible for peers to choose among policies for allocating their bandwidth. As such, the one hop protocol may also be able to incorporate a number of previously proposed reputation systems, such as Bayesian estimation [3], PPay [20], PeerTrust [19], among others [5, 7]. However, we must leave the full exploration of these issues to future work.

6 Conclusion

To deliver on their potential benefits, P2P systems need robust contribution incentives. In this paper, we have described the pitfalls undermining currently deployed incentive strategies, finding that decisions based on direct observations and local history will not suffice to overcome the performance and availability problems on which today's P2P networks falter. Our measurements motivate the design of one hop reputations, a protocol

for propagating reputations that extends the information peers have available for making servicing decisions. We propose a default policy for the use of this information, finding that for observed workloads, one hop reputations can provide wide coverage and positive, long-term contribution incentives. Through deployment on PlanetLab, we show that one hop reputations can improve short-term download performance for peers as well.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Emin Gün Sirer, for their valuable feedback. This research was partially supported by the National Science Foundation, CSR-PDOS #0720589.

References

- [1] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, October 2000.
- [2] A. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving a BitTorrent network's performance mechanisms. In *Proc. of INFOCOM*, 2006.
- [3] S. Buchegger and J.-Y. L. Boudec. A robust reputation system for P2P and mobile ad-hoc networks. In *Proc. of IPTPS*, 2004.
- [4] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. of P2P-ECON*, 2003.
- [5] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Choosing reputable servers in a P2P network. In *Proc. of WWW*, 2002.
- [6] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.
- [7] M. Gupta, P. Judge, and M. Ammar. A reputation system for peer-to-peer networks. In *Proc. of NOSSDAV*, 2003.
- [8] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Leveraging BitTorrent for end host measurements. In *Proc. of PAM*, 2007.
- [9] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigen-trust algorithm for reputation management in P2P networks. In *Proc. of WWW*, 2003.
- [10] A. Legout, N. Liogkas, E. Kohler, and L. Zhang. Clustering and sharing incentives in BitTorrent systems. In *SIGMETRICS Performance Eval. Rev.*, 2007.
- [11] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. of HotNets*, 2006.
- [12] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. Do incentives build robustness in BitTorrent? In *Proc. of NSDI*, 2007.
- [13] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. van Steen, and H. Sips. Tribler: A social-based peer-to-peer system. In *Proc. of IPTPS*, 2006.
- [14] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. of NSDI*, 2007.
- [15] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. of SIGCOMM*, 2004.
- [16] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proc. of IPTPS*, 2007.
- [17] V. Vishnumurthy, S. Chandrakumar, and E. Sirer. Karma: A secure economic framework for peer-to-peer resource sharing. In *Proc. of P2P-ECON*, 2003.
- [18] J. Wang, C. Yeo, V. Prabhakaran, and K. Ramchandran. On the role of helpers in peer-to-peer file download systems: design, analysis, and simulation. In *Proc. of IPTPS*, 2007.
- [19] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. On Knowledge and Data Engineering*, 2004.
- [20] B. Yang and H. Garcia-Molina. PPay: Micropayments for peer-to-peer systems. In *Proc. of CCS*, 2003.
- [21] Q. L. Yu. Robust incentives via multi-level tit-for-tat. In *Proc. of IPTPS*, 2006.

Ostra: Leveraging trust to thwart unwanted communication

Alan Mislove^{†‡}

Ansley Post^{†‡}

Peter Druschel[†]

Krishna P. Gummadi[†]

[†]*MPI-SWS*

[‡]*Rice University*

Abstract

Online communication media such as email, instant messaging, bulletin boards, voice-over-IP, and social networking sites allow any sender to reach potentially millions of users at near zero marginal cost. This property enables information to be exchanged freely: anyone with Internet access can publish content. Unfortunately, the same property opens the door to unwanted communication, marketing, and propaganda. Examples include email spam, Web search engine spam, inappropriately labeled content on YouTube, and unwanted contact invitations in Skype. Unwanted communication wastes one of the most valuable resources in the information age: human attention.

In this paper, we explore the use of trust relationships, such as social links, to thwart unwanted communication. Such relationships already exist in many application settings today. Our system, Ostra, bounds the total amount of unwanted communication a user can produce based on the number of trust relationships the user has, and relies on the fact that it is difficult for a user to create arbitrarily many trust relationships.

Ostra is applicable to both messaging systems such as email and content-sharing systems such as YouTube. It does not rely on automatic classification of content, does not require global user authentication, respects each recipient's idea of unwanted communication, and permits legitimate communication among parties who have not had prior contact. An evaluation based on data gathered from an online social networking site shows that Ostra effectively thwarts unwanted communication while not impeding legitimate communication.

1 Introduction

Internet-based communication systems such as email, instant messaging (IM), voice-over-IP (VoIP), online social networks, and content-sharing sites allow communication at near zero marginal cost to users. Any user with

an inexpensive Internet connection has the potential to reach millions of users by uploading content to a sharing site or by posting messages to an email list. This property has democratized content publication: anyone can publish content, and anyone interested in the content can obtain it.

Unfortunately, the same property can be abused for the purpose of unsolicited marketing, propaganda, or disruption of legitimate communication. The problem manifests itself in different forms, such as spam messages in email; search engine spam in the Web; inappropriately labeled content on sharing sites such as YouTube; and unwanted invitations in IM, VoIP, and social networking systems.

Unwanted communication wastes human attention, which is one of the most valuable resources in the information age. The noise and annoyance created by unwanted communication reduces the effectiveness of online communication media. Moreover, most current efforts to automatically suppress unwanted communication occasionally discard relevant communication, reducing the reliability of the communication medium.

Existing approaches to thwarting unwanted communication fall into three broad categories. First, one can target the unwanted communication itself, by automatically identifying such communication based on its content. Second, one can target the originators of unwanted communication, by identifying them and holding them accountable. Third, one can impose an upfront cost on senders for each communication, which may be refunded when the receiver accepts the item as wanted. Each of these approaches has certain advantages and disadvantages, which we discuss in Section 2.

In this paper, we describe a method that exploits existing trust relationships among users to impose a cost on the senders of unwanted communication in a way that avoids the limitations of existing solutions. Our system, Ostra, (i) relies on existing trust networks to connect senders and receivers via chains of pairwise trust rela-

tionships; (ii) uses a pairwise, link-based credit scheme that imposes a cost on originators of unwanted communications without requiring sender authentication or global identities; and (iii) relies on feedback from receivers to classify unwanted communication. Ostra ensures that unwanted communication strains the originator's trust relationships, even if the sender has no direct relationship with the ultimate recipient of the communication. A user who continues to send unwanted communication risks isolation and the eventual inability to communicate.

The trust relationships (or social links) that Ostra uses exist in many applications. The links can be explicit, as in online social networking sites, or implicit, as in the links formed by a set of email, IM, or VoIP users who include each other in their contact lists. Ostra can use such existing social links as long as acquiring and maintaining a relationship requires some effort. For example, it takes some effort to be included in someone's IM contact list (making that person's acquaintance); and it may take more effort to maintain that status (occasionally producing wanted communication). With respect to Ostra, this property of a social network ensures that an attacker cannot acquire and maintain arbitrarily many relationships or replace lost relationships arbitrarily quickly.

Ostra is broadly applicable. Depending on how it is deployed, it can thwart unwanted email or instant messages; unwanted invitations in IM, VoIP, or online social networks; unwanted entries or comments in blogging systems; or inappropriate and mislabeled contributions to content-sharing sites such as Flickr and YouTube.

The rest of this paper is organized as follows. Section 2 describes existing approaches to preventing unwanted communication, as well as other related work. Section 3 describes a "strawman" design that assumes strong user identities. Section 4 describes the design of Ostra and Section 5 discusses issues associated with deploying Ostra. Section 6 evaluates a prototype of Ostra on traces from an online social network and an email system. Section 7 sketches a fully decentralized design of Ostra. Finally, Section 8 concludes.

2 Related work

Unwanted communication has long been a problem in the form of email spam, and many strategies have been proposed to deal with it. However, the problem increasingly afflicts other communication media such as IM, VoIP, and social networking and content-sharing sites. In this section, we review existing approaches and describe how they relate to Ostra.

2.1 Content-based filtering

The most widespread approach to fighting unwanted communication is content-based filtering, in which recipients use heuristics and machine learning to classify communication automatically on the basis of its content. Popular examples include SpamAssassin [22] and DSPAM [7]. Content-based filters are also used for other types of unwanted communication, such as blog spam [17] and network-based security attacks [14].

Content-based filtering, however, is subject to both false positives and false negatives. False negatives — that is, when unwanted communication is classified as wanted — are a mere inconvenience. False positives [2] are a much more serious concern, because relevant messages are marked as unwanted and thus may not be received [13]. Moreover, there is a continual arms race [12] between spammers and filter developers, because the cognitive and visual capabilities of humans allow spammers to encode their message in a way that users can recognize but filtering programs have difficulty detecting.

2.2 Originator-based filtering

Another approach is to classify content by its originator's history and reputation. One such technique is whitelisting, in which each user specifies a list of other users from whom they are willing to receive content.

Whitelisting is commonly deployed in IM applications such as iChat and AIM, in VoIP systems such as Skype, and in social networking sites such as LinkedIn. In these cases, users have to be on each other's whitelists (i.e., their lists of contacts) to be able to exchange messages. To get on each other's whitelists, two users must exchange a special invitation. If the invitation is accepted, the two parties are added to each other's whitelists. If the invitation is rejected, then the inviter is added to the invitee's blacklist, which prevents the inviter from contacting the invitee again. RE [11] extends whitelists to automatically and securely include friends of friends.

To be effective, whitelisting requires that users have unique identifiers and that content can be authenticated; otherwise, it is easy for malicious users to make their communication seem to come from a whitelisted source. In most deployed email systems, messages cannot be reliably authenticated. However, secure email services, IM, VoIP services, and social networking sites can authenticate content. Whitelisting, however, cannot deal with unwanted invitations, another form of unwanted communication.

2.3 Imposing a cost on the sender

A third approach is to discourage unwanted communication by imposing a cost on the originators of either all

communication or unwanted communication. The cost can be monetary or in terms of another limited resource.

Quota- and payment-based approaches attempt to change the economics of unwanted communication by imposing a marginal cost on the transmission of an (unwanted) message.

Systems have been proposed in which senders must commit to paying a per-message fee or token before sending a digital communication [10,20,24]. These solutions attempt to model the postal service; they are based on the assumption that the cost will discourage mass distribution of unwanted messages. In some of the proposed systems, the per-message fee is charged only if the receiver classifies the messages as unwanted. This feature is important because originators of legitimate communication can still reach a large audience at low cost.

In general, deploying a decentralized email system that charges a per-message fee may require a micropayment infrastructure, which some have claimed is impractical [1, 15, 19]. Systems based on quotas do not need micropayments but still require a market for the distribution of tokens.

In challenge-response systems, the sender of a message must prove she is a human (as opposed to a computer program) before the message is delivered. Challenge-response systems can be viewed as imposing a cost on senders, where the cost is the human attention necessary to complete the challenge.

However, some automatically generated email messages (e.g., from an e-commerce site) are wanted; receiving such messages requires users to create a whitelisted email address to avoid false positives. Moreover, the need to complete a challenge may annoy and discourage some legitimate senders.

2.4 Content rating

Many content-sharing sites (e.g., YouTube [25]) use content rating. Users can indicate the level of interest, relevance, and appropriateness of a content item they have viewed. The content is then tagged with the aggregated user ratings. Content ratings can help users to identify relevant content and avoid unwanted content. These ratings can also help system administrators to identify potentially inappropriate content, which they can then inspect and possibly remove. Content rating is applicable only to one-to-many communication. Moreover, content-rating systems can be manipulated, particularly in a system with weak user identities.

2.5 Leveraging relationships

Online social relationships are used in Web-based applications for content sharing [25], socializing [9], and

professional networking [16]. LinkedIn uses implicit whitelisting of a user's friends and offers a manual introduction service based on the social network. However, none of these sites leverages the social network to enable legitimate communication automatically among users who have not had prior contact, while thwarting unwanted communication.

Trust relationships are being used in the PGP web of trust [27] to eliminate the need for a trusted certificate authority. SybilGuard [26] uses social network links to identify users with many identities (Sybils). In Ostra, we use trust relationships to ensure that a user with multiple identities cannot send additional unwanted communication, unless she also has additional relationships.

3 Ostra strawman

In this section, we describe a strawman design of Ostra. The design is appropriate for trusted, centralized communication systems in which users have strong identities (i.e., each individual user has exactly one digital identity). We discuss the basic properties of this design in the context of two-party communication (e.g., email and IM), multi-party communication (e.g., bulletin boards and mailing lists), and content-sharing sites (e.g., YouTube and Flickr). Section 4 describes a refined design that removes the need for strong identities, because such identities are difficult to obtain in practice.

3.1 Assumptions

The strawman design is based on three assumptions.

1. Each user of the communication system has exactly one unique digital identity.
2. A trusted entity observes all user actions and associates them with the identity of the user performing the action.
3. Users classify communication they receive as wanted (relevant) or unwanted (irrelevant).

Assumption 1 would require a user background check (e.g., a credit check) as part of the account creation process, to ensure that a user cannot easily create multiple identities; this assumption will be relaxed in Section 4. Assumption 2 holds whenever a service is hosted by a trusted Web site or controlled by a trusted tracker component; the trusted component requires users to log in and associates all actions with a user. We sketch a decentralized design that does not depend on this assumption in Section 7.

Producing communication can mean sending an email or chat message; adding an entry or comment to a blog;

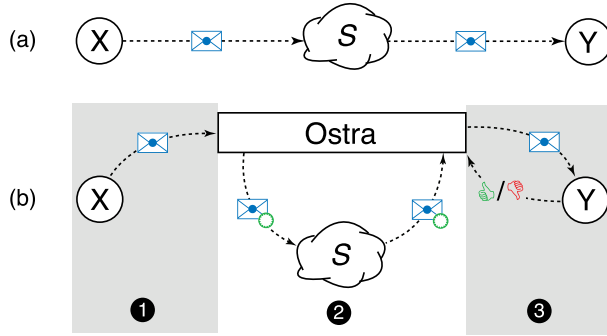


Figure 1: Diagram of (a) the original communication system S , and (b) the communication system with Ostra. The three phases of Ostra — (1) authorization, (2) transmission, and (3) classification — are shown.

sending an invitation in an IM, VoIP, or social networking system; or contributing content in a content-sharing site. Receiving communication can mean receiving a message or viewing a blog entry, comment, or search result.

Typically, a user considers communication unwanted if she feels the content was not worth the attention. A user considers a blog entry, comment, or content object as unwanted if she considers the object to be inappropriate for the venue (e.g., site, group, or blog space) it was placed in or to have inappropriate search tags, causing the object to appear in response to an unrelated search.

3.2 System model

Figure 1 shows how Ostra interacts with a given communication system S . Ostra is a separate module that runs alongside the existing communication system. With Ostra, communication consists of three phases.

Authorization: When a sender wishes to produce a communication, she first passes the communication to Ostra. Ostra then issues a token specific to the sender, recipient, and communication. If the sender has previously sent too much unwanted communication, Ostra refuses to issue such a token and rejects the communication.

Transmission: Ostra attaches the token to the communication and transmits it using the existing communication mechanism. On the receiving side, Ostra accepts the communication if the token is valid. The communication is then provided to the recipient. Note that Ostra is not involved in the actual transmission of the communication.

Classification: The recipient classifies the communication as either wanted or unwanted, according to her personal preferences. This feedback is then provided to Os-

tra. Finally, Ostra makes this feedback available to the sender.

3.3 User credit

Ostra uses credits to determine whether a token can be issued. Each user is assigned a credit balance, B , with an initial value of 0. Ostra also maintains a per-user balance range $[L, U]$, with $L \leq 0 \leq U$, which limits the range of the user's credit balance (i.e., $L \leq B \leq U$ at all times). We denote the balance and balance range for a single user as B_L^U . For example, if a user's state is 3_{-5}^{+6} , the user's current credit balance is 3, and it can range between -5 and 6.

When a token is issued, Ostra requires the sender to reserve a credit and the receiver to reserve a place holder for this credit in their respective credit balances. To make these reservations, the sender's L is raised by one, and the receiver's U is lowered by one. If these adjustments would cause either the sender's or the receiver's credit balance to exceed the balance range, Ostra refuses to issue the token; otherwise, the token is issued. When the communication is classified by the receiver, the range adjustments are undone. If the communication is marked as unwanted, one credit is transferred from the sender to the receiver.

Let us consider an example in which both the sender's and the receiver's initial balances are 0_{-3}^{+3} . When the token is issued, the sender's balance changes to 0_{-2}^{+3} , and the receiver's balance changes to 0_{-3}^{+2} , representing the credit reservation. Let us assume that the communication is classified as unwanted. In this case, a credit is transferred from the sender to the receiver; the receiver's balance becomes 1_{-3}^{+3} , and the sender's becomes -1_{-3}^{+3} .

This algorithm has several desirable properties. It limits the amount of unwanted communication a sender can produce. At the same time, it allows an arbitrary amount of wanted communication. The algorithm limits the number of tokens a user can acquire before any of the associated communication is classified; thus, it limits the total amount of potentially unwanted communication a user can produce. Finally, the algorithm limits the number of tokens that can be issued for a specific recipient before that recipient classifies any of the associated communication; thus, an inactive or lazy user cannot cause senders to reserve a large number of credits, which would be bound until the communication were classified.

3.4 Credit adjustments

Several issues, however, remain with the algorithm described so far. When a user's credit balance reaches one of her credit bounds, she is, in effect, banned from producing (in the case of the lower bound) or receiving (in

the case of the upper bound) any further communication. What can cause a legitimate user's credit balance to reach her bounds? Note that on the one hand, a user who receives unwanted communication earns credit. On the other hand, even a well-intentioned user may occasionally send communication to a recipient who considers it unwanted and therefore lose credit. Across all users, these effects balance out. However, unless a user, on average, receives precisely the same amount of unwanted communication as she generates, her credit balance will eventually reach one of her bounds. As a result, legitimate users can find themselves unable to communicate.

To address this problem, credit balances in Ostra decay towards 0 at a constant rate d with $0 \leq d \leq 1$. For example, Ostra may be configured so that each day, any outstanding credit (whether positive or negative) decays by 10%. This decay allows an imbalance between the credit earned and the credit lost by a user. The choice of d must be high enough to cover the expected imbalance but low enough to prevent considerable amounts of intentional unwanted communication. As we show as part of Ostra's evaluation, a small value of d is sufficient to ensure that most legitimate users never exceed their credit range.

With this refinement, Ostra ensures that each user can produce unwanted communication at a rate of at most

$$d * L + S$$

where S is the rate at which the user receives communication that she marks as unwanted.

A denial of service attack is, however, still possible. Colluding malicious users can inundate a victim with large amounts of unwanted communication, causing the victim to acquire too much credit to receive any additional communication. For these users, the rate of decay may be too low to ensure that they do not exceed their credit balances. To prevent such attacks, we introduce a special account, C , that is not owned by any user and has no upper bound. Users with too much credit can transfer credit into C , thereby enabling them to receive further communication. Note that the credit transferred into C is subject to the usual credit decay, so the total amount of credit available to active user accounts does not diminish over time. Additionally, users can only deposit credit into C ; no withdrawals are allowed.

Finally, there is an issue with communication failures (e.g., dropped messages) and users who are offline for extended periods. Both may cause the sender to reserve a credit indefinitely, because the receiver does not classify the communication. The credit decay does not help in this situation, because the decay affects only the credit balance and not the credit bounds. Therefore, Ostra uses a timeout T , which is typically on the order of days. If a communication has not been classified by the re-

Operation	Net Change in System Credit
User joins system	0, as user's initial credit balance is 0
Wanted comm. sent	0, as no credit is exchanged
Unwanted comm. sent	0, as credit is transferred between users
Daily credit decay	0, as total credit was 0 before decay

Table 1: Operations in Ostra, and their effect on the total system credit. No operation alters the sum of credit balances.

ceiver after T , the credit bounds are automatically reset as though the destination had classified the communication as wanted. This feature has the added benefit that it enables receivers to plausibly deny receipt of communication. A receiver can choose not to classify some communication, thus concealing its receipt.

3.5 Properties

Ostra's system of credit balances observes the following invariant:

At all times, the sum of all credit balances is 0

The conservation of credit follows from the fact that (i) users have an initial zero balance when joining the system, (ii) all operations transfer credit among users, and (iii) credit decay affects positive and negative credit at the same rate. Table 1 details how each operation leaves the overall credit balance unchanged. Thus, credit can be neither created nor destroyed. Malicious, colluding users can pass credits only between themselves; they cannot create additional credit or destroy credit. The amount of unwanted communication that such users can produce is the same as the sum of what they can produce individually.

We have already shown that each user can produce unwanted communication at a rate of no more than $d * L + S$. We now characterize the amount of unwanted subset of the user population can produce. Let us examine a group of users F . Owing to the conservation of credit, users in this group cannot conspire to create credit; they can only push credit between themselves. Thus, the users in F can send unwanted communication to users not in F at a maximal rate of

$$|F| * d * L + S_F$$

where S_F is that rate at which users in F (in aggregate) receive communication from users not in F that they mark as unwanted.

The implication of the above analysis is that we can characterize the total amount of unwanted communication that non-malicious users can receive. Let us partition the user population into two groups: group G are "good" users, who rarely send unwanted communication, and

	Action	Cost	Reward
Sending	Send wanted communication	1 credit	
	Send unwanted communication		
Classifying	Classify as wanted	Sender likely to send more Sender unlikely to send more	Sender likely to send more 1 credit, throttle sender
	Classify as unwanted		
	Misclassify as wanted		1 credit
	Misclassify as unwanted		
Abuse	Don't use token	Ties up credit for T	
	Don't classify	Ties up credit for T	
	Drop incoming communication (§7)	1 credit	

Table 2: Incentives for users of Ostra. Users are incentivized to send only wanted communication, to classify communication correctly, and to classify received communication promptly. Marking an incoming communication as unwanted has the effect of discouraging the sender from sending additional communication, as the sender is informed of this and loses credit. Alternatively, marking an incoming communication as wanted costs the sender nothing, allowing the sender to send future communication with increased confidence.

group M are “malicious” users, who frequently send unwanted communication. Now, the maximal rate at which G can receive unwanted communication from M is

$$|M| * d * L + S_M$$

which implies that, on average, each user in G can receive unwanted communication at a rate of

$$\frac{|M|}{|G|} * d * L + \frac{S_M}{|G|}$$

However, we expect S_M to be small as users in G rarely send unwanted communication. Thus, the rate of receiving unwanted communication is dominated by static system parameters and by the ratio between the number of good and malicious users. Moreover, this analysis holds regardless of the amount of good communication that the malicious users produce.

Finally, Ostra has an incentive structure that discourages bad behavior and rewards good behavior. Table 2 shows a list of possible user actions and their costs and rewards.

3.6 Multi-party communication

Next, we show how the design can be used to support moderated multi-party communication, including mailing lists and content-sharing sites. The existing design generalizes naturally to small groups in which all members know each other. In this case, communication occurs as a series of pairwise events between the originator and each of the remaining group members.

In moderated groups, which are usually larger, a moderator decides on behalf of the list members if communication submitted to the group is appropriate. In this case, Ostra works exactly as in the two-party case, except that the moderator receives and classifies the communication on behalf of all members of the group.

Thus, only the moderator’s attention is wasted by unwanted communication, and the cost of producing unwanted communication is the same as in the two-party case. However, an overloaded moderator may choose to increase the number of credits required to send to the group, to mitigate her load by discouraging inappropriate submissions.

Large content-sharing sites usually have content-rating systems or other methods for flagging content as inappropriate. Ostra could be applied, for instance, to thwart the submission of mislabeled videos in YouTube, by taking advantage of the existing “flag as inappropriate” mechanism. When a user’s video is flagged as inappropriate, it is reviewed by a YouTube employee; if it is found to be mislabeled, the submission is classified as unwanted for the purposes of Ostra.

Extending Ostra to work with unmoderated multi-party communication systems is the subject of ongoing work but is beyond the scope of this paper.

4 Ostra design

The strawman design described in the previous section requires strong user identities: that is, each individual user is guaranteed to have at most one unique digital identity. Such identities are not practical in many applications, as they require a background check as part of the account creation process. Such checks may not be accepted by users, and as far as we know, few services that require such a strong background check have been widely adopted on the Internet.

In this section, we refine the design of Ostra so that it does not require strong user identities. It is assumed that the communication system ensures that each identity is unique, but an individual user may sign up multiple times and use the system under different identities at different times. Our refined design leverages trust relationships to preserve Ostra’s properties despite the lack of strong

user identities. We still assume that a trusted entity such as a Web site hosts the communication service and runs Ostra. Later, in Section 7, we sketch out how Ostra could be applied to decentralized services.

The refined design of Ostra replaces the per-user credit balances with balances that are instead associated with the links among users in a trust network. We show that this mapping preserves the key properties of the strawman design, even though Ostra no longer depends on strong identities. We begin by defining a trust network and then describe how Ostra works with weak identities.

4.1 Trust networks

A trust network is a graph $G = (V, E)$, where V is the set of user identifiers and E represents undirected links between user identifiers who have a trust relationship. Examples of trust networks are the user graph of an email system (where V is the set of email addresses and E is the set of email contacts) and online social networks (where V is the set of accounts and E is the set of friends). For convenience, we shall refer to two users connected by an edge in the trust network as friends.

For the purposes of Ostra, a trust network must have the property that there is a non-trivial cost for initiating and maintaining links in the network. As a result, users in the trust network cannot acquire new relationships arbitrarily fast and cannot maintain an arbitrarily large number of relationships. We do not make any assumptions about the nature or the degree of trust associated with a relationship.

Finally, the trust network must be connected, meaning that there is a path of trust links between any two user identities in the network. Previous studies [4, 18] have shown that the user graphs in existing social networks tend to be dominated by a single large component, implying that the networks are largely connected.

Ostra assumes that the users of a communication system are connected by a trust network and that Ostra has a complete view of this network.

4.2 Link credit

Because a user may have multiple identities, we can no longer associate a separate credit balance with each identity. Otherwise, a malicious user could gain additional credit and send arbitrary amounts of unwanted communication simply by creating more identities. Instead, Ostra leverages the cost of forming new links in trust networks to enforce a bound on each user.

Specifically, each link in the trust network is assigned a link credit balance B , with an initial value of 0, and a link balance range $[L, U]$, with $L \leq 0 \leq U$ and $L \leq B \leq U$. These are analogous to the user credit

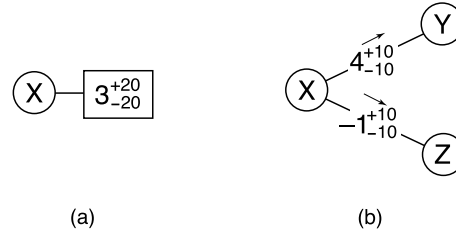


Figure 2: Mapping from (a) per-user credits to (b) per-link credits.

balance and range in the original design. We denote the balance and balance range for a link $X \leftrightarrow Y$ from X 's perspective as $B_{L}^{X \rightarrow Y}$. For example, if the link has the state 3_{-5}^{+6} , then X is currently owed 3 credits by Y , and the balance can range between -5 and 6 .

The link balance represents the credit state between the user identities connected by the link. Ostra uses this balance to decide whether to issue tokens. It is important to note that the credit balance is symmetric. For example, if the link balance on the $X \leftrightarrow Y$ link is 1_{-2}^{+3} , then X is owed one credit by Y , or, from Y 's perspective, Y owes X one credit (the latter can be denoted -1_{-3}^{+2}).

We map the user credit balance in the strawman design to a set of link credit balances on the user's adjacent links in the trust network. For example, as shown in Figure 2, if a user has two links in the trust network, the user's original credit balance is replaced with two separate credit balances, one on each link. However, we cannot compute a user balance by taking the sum of the link balances – in fact, the concept of a user balance is no longer useful because a user can create many identities and establish links between them. Instead, we introduce a new mechanism for credit transfer that uses link balances, rather than user balances, to bound the amount of unwanted communication that users can send.

We now describe this mechanism for transferring credits. For simplicity, we first describe the case of communication between users who are friends in the trust network. We then generalize the credit transfer mechanism to the case in which two arbitrary users wish to communicate.

4.2.1 Communication among friends

As in the strawman design, a user who wishes to send communication needs to obtain a token during the authorization phase. For example, a user X may request to send communication to another user Y , a friend of X 's. Ostra determines whether transferring this credit would violate the link balance range on the $X \leftrightarrow Y$ link, and if

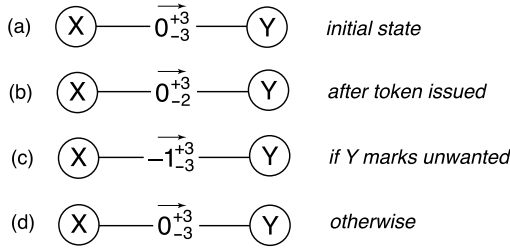


Figure 3: Link state when X sends communication to friend Y . The state of the link balance and range is shown (a) before the token is issued, (b) after the token is issued, (c) if Y marks the communication as unwanted, and (d) if Y marks the communication as wanted or if the timeout occurs.

not, it issues a signed token. The token is then included in X 's communication to user Y .

As in the strawman design, Ostra allows users to have multiple outstanding tokens by reserving credits for each potential transfer. In the example in the previous paragraph, Ostra raises the lower bound for the $X \leftrightarrow Y$ link by one. This single adjustment has the effect of raising X 's lower bound and lowering Y 's upper bound, because the lower bound on the $X \leftrightarrow Y$ link can be viewed as the upper bound on the $Y \leftrightarrow X$ link. Figure 3 shows the state of the $X \leftrightarrow Y$ link during each stage of the transaction. By adjusting the balance this way for outstanding tokens, Ostra ensures that the link balance remains within its range regardless of how the pending communication events are classified.

Later, in the classification stage, user Y provides Ostra with the token and the decision whether X 's communication was wanted. The balance range adjustment that was performed in the authorization phase is then undone. Moreover, if Y reports that the communication was unwanted, Ostra adjusts the balance on the $X \leftrightarrow Y$ link by subtracting one, thereby transferring a credit from X to Y . Thus, if the previous state of the link was $\begin{smallmatrix} X \rightarrow Y \\ 0 & +3 \\ & -3 \end{smallmatrix}$, the final state would be $\begin{smallmatrix} X \rightarrow Y \\ -1 & +3 \\ & -3 \end{smallmatrix}$, meaning X owes Y one credit. Finally, Ostra automatically cancels the token after a specified timeout T .

4.2.2 Communication among non-friends

So far, we have considered the case of sending communication between two friends in the trust network. In this section, we describe how Ostra can be used for communication between any pair of users.

When a user X wishes to send communication to a non-friend Z , Ostra finds a path consisting of trust links between X and Z . For example, such a path might be $X \leftrightarrow Y \leftrightarrow Z$, where X and Y are friends in the trust

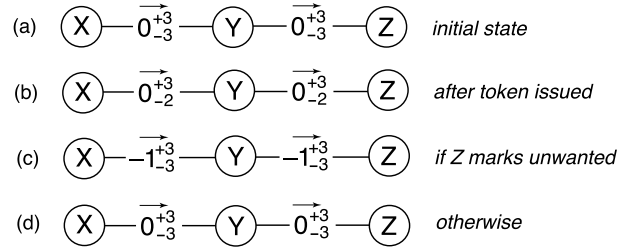


Figure 4: Link state when X sends communication to non-friend Z is shown (a) before the token is issued, (b) after the token is issued, (c) if Z marks the communication as unwanted, and (d) if Z marks the communication as wanted or if the timeout occurs.

network, and Y and Z are also friends. When this path is found, the range bounds are adjusted as before, but this occurs on every link in the path. For example, if X wishes to send communication to Z , Ostra would raise the lower bound of both the $X \leftrightarrow Y$ and the $Y \leftrightarrow Z$ links by one. Figure 4 shows a diagram of this procedure. If this adjustment can be done without violating any link ranges, Ostra issues a token to X .

Similar to the transfer between friends, the token is then attached to X 's communication to Z . Later, in the classification stage, Z provides Ostra with the token and the decision whether the communication was wanted. Now, the range adjustments on all the links along the path are undone. If the communication was unwanted, the credit is transferred along every link of the path; Figure 4 (c) shows the result of this transfer.

It is worth noting that the intermediate users along the path are largely indifferent to the outcome of the transfer, as any credit transfer will leave them with no net change. For example, consider the scenarios shown in Figure 4 (c) and (d). In either case, the total amount of credit that intermediate user Y has with all her friends is the same regardless of the outcome. If Z marks the communication as unwanted, as shown in Figure 4(c), Y owes a credit to Z , but X now owes a credit to Y . Ostra allows users to transfer credits along trust paths such that intermediate users along the path are indifferent to the outcome.

4.2.3 Generalization of Ostra strawman

One can show that Ostra generalizes the strawman design from the previous section. Recall the account C that is owned by the trusted site. Now, we construct a trust network in which each user has a single link to C , with the link balance and balance range equal to their user balance and balance range in the strawman design. Ostra with such a trust network has the same properties as the strawman design. To see this, note that sending commu-

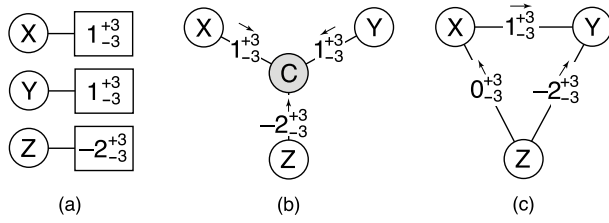


Figure 5: Generalization of per-user credit accounting to per-link credit accounting. Ostra with per-user credit (shown in (a)) can be expressed as per-link credit over a star topology (shown in (b)), with the central site C as the hub. The addition of links (shown in (c)) does not change the properties.

nication from X to Y requires raising the lower bound on the $X \leftrightarrow C$ link and lowering the upper bound on the $Y \leftrightarrow C$ link, which is equivalent to adjusting X 's and Y 's user balance ranges in the same manner. Figure 5 (b) shows an example of this generalization for the specific set of user accounts in Figure 5 (a).

More importantly, Ostra preserves the conservation of credit that was present in the strawman system. This can be derived from the fact that credit is associated with links instead of users. Any credit in Ostra is naturally paired with a corresponding debt: for example, if the state of a link is -1_{-2}^{+3} , then X owes Y one credit, but Y is owed a credit by X . Thus, all outstanding credit is balanced by outstanding debt, implying that credit cannot be created or destroyed.

The conservation of credit holds for each link independently, and is therefore independent of the trust network topology (Figure 5 (c) shows an example of a trust network with a different topology). As a result, the analysis of the strawman system in Section 3.5 applies to the full version of Ostra. For example, malicious, colluding users cannot conspire to manufacture credit; the amount of unwanted communication that such users can produce together is the sum of what they can produce independently.

4.3 Security properties

We now discuss the security properties of Ostra's refined design in detail. Ostra's threat model assumes that malicious users have two goals: sending large amounts of unwanted communication, and preventing other users from being able to send communication successfully. Strategies for trying to send additional unwanted communication include signing up for multiple accounts and creating links between these accounts, as well as conspiring with other malicious users. Strategies for trying to prevent other users from communicating include targeting a

specific user by sending large amounts of unwanted communication and attempting to exhaust credit on specific links in the trust network. In this section, we describe how Ostra handles these threats.

4.3.1 Multiple identities

One concern is whether users who create multiple identities (known as Sybils [6]) can send additional unwanted communication. Ostra naturally prevents such users from gaining additional credit.

To send unwanted communication to another user, a user must eventually use one of her "real" links to a different user, which has the same effect as if the user only had a single identity. To see this, assume a user with a set of multiple identities $M = \{M_1, M_2, \dots, M_n\}$ is sending to a different user U . Now, regardless of how the links between the identities in M are allocated, any path between M_i and U must contain a link $M_j \leftrightarrow V$, where $V \notin M$. If this property does not hold, then $U \in M$, which is a contradiction.

Thus, using per-link balances has the effect that the total credit available to a user no longer depends on the number of identities a user has. Instead, the credit available depends on the number of links the user has to other users. Figure 6 shows a diagram of how Ostra prevents users with multiple identities from sending additional unwanted communication.

Ostra allows users to create as many identities as they wish but ensures that they cannot send additional unwanted communication by doing so. Malicious users may attempt to use multiple Sybil identities to create multiple links to a single user. Although they may succeed occasionally, these links require effort to maintain and the malicious user, therefore, cannot create an unbounded number of them.

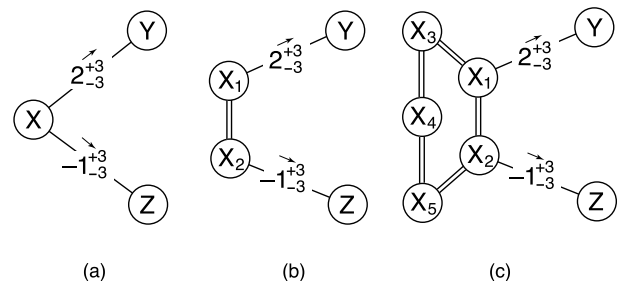


Figure 6: Diagram of how Ostra handles various attacks: (a) a normal user, (b) multiple identities, and (c) a network of Sybils. The total amount of credit available to the user is the same.

4.3.2 Targeting users

Another concern is whether malicious users could collectively send a large amount of unwanted communication to a user, thus providing this victim with too much credit to receive any additional messages. This attack is possible when the attacking users collectively have more links to legitimate users than the victim, as exhausting the credit on one of the victim's links requires the malicious users exhaust the credit on one of their own links.

However, the victim has a simple way out by forgiving some of the debt on one of her links. If a user finds that she has too much credit on all of her links, she can forgive a small amount of debt from one of her friends. This is the same mechanism as transferring credit to the overflow account (C) described in Section 3. To see this equivalence, consider the star-topology trust network constructed in Section 4.2.3. In that case, a user transferring credit to the overflow account is essentially forgiving debt on their only link (to C). This mechanism does not allow malicious users to send additional unwanted communication to the victim, as the victim only forgives debt to her direct friend (i.e., the victim's friend does not repeat the process).

4.3.3 Targeting links

One final concern is whether malicious users could prevent large numbers of innocent users from communicating with each other by exhausting the credit on certain links in the trust network. If successful, such an attack could prevent a group of users from sending to the rest of the user population.

To exhaust the credit on specific links, attacking users would need both knowledge of the trust network topology and some control over trust path selection. Because the path selection is performed by the trusted site, the attacking users have the choice of only the destination and not the path. Even if we assume a powerful attacker who has control over the path selection, the trust network would need to have a topology that is susceptible to such an attack. For example, a barbell topology would be susceptible, as the link connecting the two halves of the network could be exhausted.

Analysis of current online social networks (which are typical trust networks) shows that these have a very dense core [18]. We show in Section 6 that the structure of these networks makes it unlikely that such an attack would succeed on a large scale.

5 Discussion

In this section, we discuss some issues associated with deploying Ostra.

5.1 Joining Ostra

Fundamentally, Ostra exploits the trust relationships in an existing social network of users to thwart unwanted communication. As a result, users are expected to acquire and maintain a certain number of social links to be able to communicate.

To join Ostra, a new user must be introduced to the system by an existing Ostra user. Requiring this form of introduction ensures that the trust network among users is connected and that each new user has at least one trust link. Thus, Ostra can be used only in conjunction with a "invitation-only" social network.

Users with few links in the trust network are more susceptible to credit exhaustion (whether accidental or malicious). Thus, there is an incentive for users to obtain and maintain a sufficient number of trust links. Establishing additional links can be done via the communication system after the user has joined Ostra. Link invitations are treated as normal messages, so users who attempt to send unwanted link invitations are blocked in the same manner as users who send other forms of unwanted communication.

5.2 Content classification

Ostra requires that recipients classify incoming communication as either wanted or unwanted. Providing explicit feedback is a slight burden on the user, but it may be a small price to pay for a system that responds to each user's preferences and is free of the misclassifications that are common in current content-based filtering systems [2]. Moreover, the feedback can often be derived implicitly from a user's actions; for instance, deleting a message probably indicates that the message was unwanted, whereas archiving or replying to the message strongly indicates that it was wanted.

As an optimization in message-based communication systems, a user could maintain a whitelist indicating users from whom communication is immediately and unconditionally classified as wanted. In this case, Ostra would need to operate only among users who are not on each other's whitelists.

5.3 Parameter settings

Ostra limits the amount of pending communication that a user can have, where a pending item of communication is one that was generated by the user but not yet classified by the receiver. In Section 6, we show that Ostra's design parameters (L , U , and d) can be chosen such that most legitimate users are not affected by the rate limit, while the amount of unwanted communication is still kept very low.

The L parameter controls the number of unclassified items of communication a user can have at any one time. A large L allows many outstanding messages but also admits the possibility that a considerable amount of this outstanding communication would be unwanted. In contrast, an L close to 0 ensures that very little unwanted communication is received, at the cost of potentially rate-limiting legitimate senders. The d parameter represents the rate at which users who have sent unwanted communication in the past are “forgiven”. Setting d too high allows additional unwanted communication, whereas setting it too low may unduly punish senders who have inadvertently sent unwanted communication in the past. In the Section 6, we show that the conservative settings of $L=3$ and $d=10\%$ per day provide a good trade-off in practice.

5.4 Compromised user accounts

If a user’s account password is compromised, the attacker can cause the user to run out of credit by sending unwanted communication. However, the amount of unwanted communication is still subject to the same limits that apply to any individual user. Moreover, a user would quickly detect that her account has been compromised, because she would find herself unable to generate communication.

6 Evaluation

In this section, we present an experimental evaluation of our Ostra prototype. Using data from a real online social network and an email trace from our institute, we show how Ostra can effectively block users from sending large amounts of unwanted communication.

6.1 Experimental trust network

To evaluate Ostra, we used a large, measured subset [18] of the social network found in the video-sharing Web site YouTube [25]. We extracted the largest strongly connected component consisting of symmetric links from the YouTube graph, which resulted in a network with 446,181 users and 1,728,938 symmetric links.

Strictly speaking, the YouTube social network does not meet Ostra’s requirements, because there is no significant cost for creating and maintaining a link. Unfortunately, trust-based social networks that do meet Ostra’s requirements cannot be easily obtained due to privacy restrictions. For instance, in the LinkedIn [16] professional networking site, users “vouch” for each other; link formation requires the consent of both parties and users tend to refuse to accept invitations from people they do not

know and trust. But, unlike YouTube, it is not possible to crawl the LinkedIn network.

However, we were able to obtain the degree distribution of users in the LinkedIn network. We found that both YouTube and LinkedIn degree distributions follow the power-law with similar coefficients. We used maximum-likelihood testing to calculate the coefficients of the YouTube and LinkedIn graphs, and found them to be 1.66 and 1.58 (the resultant Kolmogorov-Smirnov goodness-of-fit metrics were 0.12 and 0.05, suggesting a good fit). This result, along with the previously observed similarity in online social networks’ structure [18], leads us to expect that the overall structure of the YouTube network is similar to trust-based social networks like LinkedIn.

Despite their structural similarity, the YouTube social network differs from the LinkedIn trust network in one important aspect: some users in YouTube collect many links (one user had a degree of over 20,000!). The maximum degree of users in actual trust-based social networks tends to be much smaller. Anthropological studies [8] have shown that the average number of relationships a human can actively maintain in the real world is about 150 to 200. Because the amount of unwanted communication a user can send in Ostra is proportional to her degree in the trust network, the results of our YouTube-based evaluation may understate the performance of Ostra on a real trust-based network.

6.2 Experimental traffic workload

We were unable to obtain a communication trace of the same scale as the social network we use. Therefore, we had to make some assumptions about the likely communication pattern within the social network. We expect that users communicate with nearby users much more often than they communicate with users who are far away in the social network. To validate this hypothesis, we collected an email trace from our organization, consisting of two academic research institutes with approximately 200 researchers. Our anonymized email trace contains all messages sent and received by the mail servers for 100 days, and the anonymized addresses in the trace are flagged as internal or external addresses.

Similar to previous studies [5, 21], we extracted a social network from the email data by examining the messages sent between internal users. Specifically, we created a symmetric link between users who sent at least three emails to each other. We filtered out accounts that were not owned by actual users (e.g., helpdesk tickets and mailing lists), resulting in a large strongly connected component containing 150 users and covering 13,978 emails.

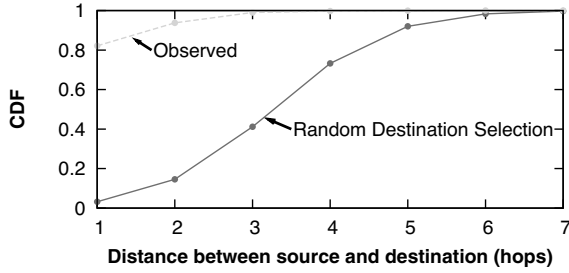


Figure 7: Cumulative distribution (CDF) of distance between sender and receiver for our email trace. The observed data show a strong bias toward proximity when compared to randomly selected destinations.

We then examined the social network distance between sender and receiver for all messages sent between these 150 users. Figure 7 compares the resulting distance distribution with one that would result had the senders selected random destinations. We found that the selection of senders had a very strong proximity bias: over 93% of all messages were sent to either a friend or a friend of a friend, compared to the expected 14% if the senders were chosen randomly. Thus, we expect that in practice, most communication in Ostra is directed to nearby users, significantly reducing the average path lengths in the trust network.

6.3 Setting parameters

We also used the email trace to determine the appropriate settings for the Ostra parameters L and U . To do this, we examined the rate at which users sent and received messages. The trace contains 50,864 transmitted messages (an average of 3.39 messages sent per user per day) and 1,003,819 received messages (an average of 66.9 messages received per user per day). The system administrators estimated that the incoming messages in the email trace consisted of approximately 95% junk mail. Clearly, most of these receptions would not occur in an actual Ostra deployment. However, we could not access the spam filter's per-message junk mail tags, so we randomly removed 95% of the incoming messages as junk.

To determine how often a given setting of L and U would affect Ostra, we simulated how messages in the email trace would be delayed due to the credit bounds. We ran two experiments with different assumptions about the average delay between the time when a message arrives and the time when the receiving user classifies the message. We first simulated casual email users who classify messages after six hours, and we then simulated heavy email users who classify messages after two hours.

	Avg. classific. delay (h)	Fraction delayed	Delay (h)		
			Avg.	Med.	Max.
Send	2	0.38%	2.2	1.9	7.6
	6	0.57%	6.1	5.3	23.6
Recv	2	1.3%	4.1	3.2	13.2
	6	1.3%	16.6	14.7	48.6

Table 3: Message delays in sending (Send) and receiving (Recv) with $L=3$ and $U=3$. The delays are shown for heavy email users (2 hour average classification delay) and casual email users (6 hour average classification delay).

Table 3 presents the results of these two experiments with $L=3$ and $U=3$. We found that messages are rarely delayed (less than 1.5% of the time in all cases), and that the average delay is on the order of a few hours. We also found that the delays for receiving messages are more significant than the delays for sending messages. We believe this is an artifact of our methodology. Over 98% of the delayed messages were received by just 3 users. In practice, it is likely that these users (who receive a high volume of relevant email) check and classify their email very frequently. This effect would reduce the frequency and magnitude of delays, but our simulation does not account for it.

6.4 Effectiveness of Ostra

In this section, we simulate deployments of Ostra in a message-based system (such as the messaging service on Flickr) and in a content-sharing system (such as YouTube). We evaluate Ostra under three traffic workloads: *Random*, where users select destinations randomly; *Proximity*, where users select destinations with the distribution that was observed in Section 6.2; and *YouTube*, where users send to a single YouTube account in the network. We show that in all cases, Ostra effectively bounds the rate at which malicious users can send unwanted communication while not impeding wanted communication.

6.4.1 Expected performance

Ostra limits the amount of unwanted communication that can be sent. A single user can send unwanted communication at a rate of at most $d * L * D + S$, where D is the degree of the user. Thus, the rate at which a malicious user can send unwanted communication is in direct proportion to her degree. As the d or L parameters are increased, we expect the rate of unwanted communication to increase accordingly. Additionally, as the proportion of malicious users in the network increases, we expect the overall rate of unwanted messages to increase.

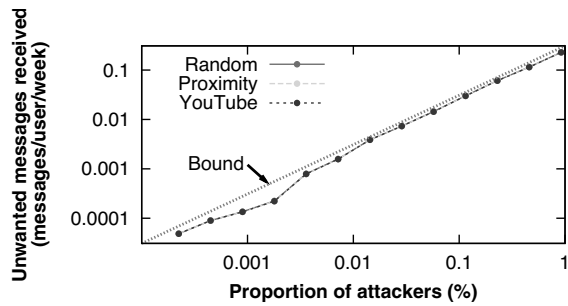


Figure 8: Amount of unwanted communication received by good users as the number of attackers is varied. As the number of attackers is increased, the number of unwanted messages delivered scales linearly.

6.4.2 Preventing unwanted communication

In this section we verify experimentally that Ostra performs as described in Section 6.4.1. Unless otherwise noted, the experiments were run with 512 randomly chosen attackers (approximately 0.1% of the population), $L=-3$, $U=3$, and $d=10\%$ per day. Each good user sent 2 messages and each attacker sent 500 messages.

To evaluate Ostra in the context of a content-sharing site, we modeled Ostra working in conjunction with YouTube. For these experiments, we configured the network so that uploading a video involves sending a message via Ostra to a single ‘YouTube’ account in the network. An existing, well-connected user (1,376 links) in the core of the network was selected to represent this account.

We first show that the rate at which users receive unwanted communication varies with the number of attacking users. In Figure 8, we present the results of experiments in which we vary the number of attackers in the network between 1 and 4,096 users (0.0002% to 1% of the network). We examine the rate at which unwanted messages were received by non-attacking users, along with the expected bound derived from the equations in Section 6.4.1.

As can be seen in Figure 8, Ostra effectively bounds the number of unwanted messages in proportion to the fraction of users who send unwanted communication. Even with 1% of the network sending unwanted messages, each legitimate user receives only 0.22 unwanted messages per week, translating to approximately 12 unwanted messages per year.

Next, we explore Ostra’s sensitivity to system parameter settings and other conditions. Important parameters in Ostra are the credit bounds L and U for each link. If these bounds are set too high, attackers can send many messages before being cut off. However, if these bounds are set too low, a legitimate user could be temporarily prevented from sending messages. Figure 9 shows how

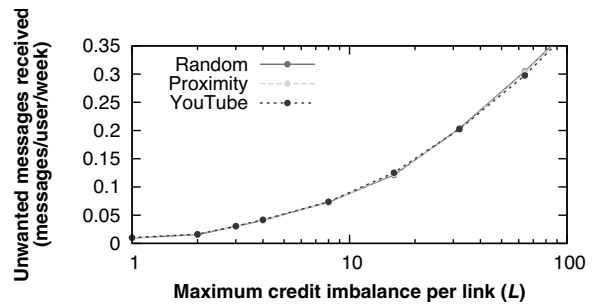


Figure 9: Amount of unwanted communication received by good users as the maximum credit imbalance per link is varied.

the rate of unwanted message delivery is affected by the maximal credit imbalance across a link. As the maximum allowed imbalance increases, the amount of unwanted communication received by good users increases, as expected.

Finally, we examine the sensitivity of Ostra to the false positive rate of legitimate users’ message classification. In other words, if users incorrectly mark other good users’ messages as unwanted, how often are users blocked from sending message? We show how this probability of false classification affects the proportion of messages that cannot be sent in Figure 10. As can be seen, even a high false positive rate of 30% results in only a few blocked messages. This resiliency results from the rich connectivity of the social network (i.e., if one link becomes blocked, the users can route through other friends), and the fact that the false positive rate affects all users equally.

In the case of the content-sharing site, because all paths intersect, good users are blocked more quickly as the amount of content that is marked as unwanted increases. For example, when the false classification rate is 64%, about 40% of messages cannot be sent. However, it seems very unlikely that the moderator of a sharing site would misclassify content at such a high rate.

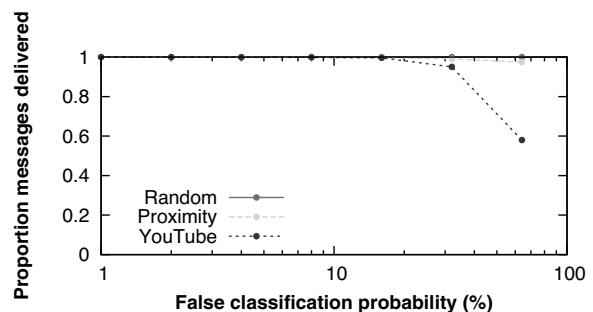


Figure 10: Proportion of messages delivered versus false classification probability for wanted messages.

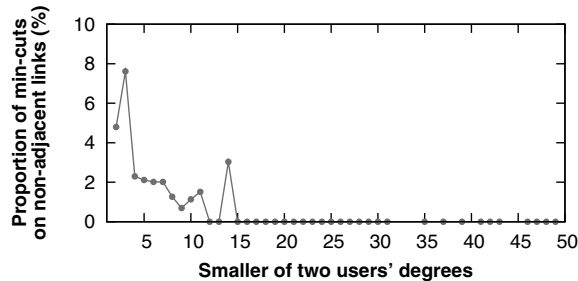


Figure 11: Proportion of 3,000 random user pairs for which the min-cut was not adjacent to one of the users, as a function of the lower of the two users' degrees. The fraction decreases as the users become well-connected, suggesting that a trust network with well-connected users is not vulnerable to link attacks.

6.4.3 Resilience to link attacks

In a potential security attack discussed in Section 4, malicious users attempt to exhaust credit on a set of links inside the trust network, i.e., links other than the attackers' adjacent links. If successful, this attack could disrupt communication for innocent users. To evaluate whether a real-world social network is susceptible to this attack, we performed a min-cut analysis of the YouTube social network.

Assuming uniform link weights of one, we calculated the min-cuts between 3,000 randomly selected pairs of users. (A min-cut is a minimal set of links that, if removed, partitions two users; note that several such cuts can occur between two users.) We then looked for cases in which the set of links involved in a min-cut for a given pair of users differed from the set of links adjacent to either one of the two users. Such a min-cut could be the target of an attack, because the attackers could exhaust credit on this set of links before they exhaust the credit on their own links.

Figure 11 plots the proportion of user pairs for which the min-cut was not adjacent to one of the users, as a function of the lower of the two users' degrees. The results suggest that vulnerable links inside the network occur rarely, and that their frequency decreases with the degree of user connectivity. Therefore, the better connected users are in the trust network, the more robust the network is to link attacks. Because users in Ostra already have an incentive to maintain a certain number of links for other reasons, one would expect that a real Ostra trust network would not be vulnerable to link attacks.

7 Decentralizing Ostra

The design of Ostra we have described so far assumes the existence of a trusted, centralized component that main-

tains the trust network and credit state. This design is suitable for centralized communication systems, such as those hosted by a Web site. Peer-to-peer communication systems with a centralized "tracker" component can also use this design. However, completely decentralized systems like SMTP-based email cannot use it. In this section, we briefly sketch out a design of Ostra that works without any trusted, centralized components.

7.1 Overview

In the absence of a trusted, centralized entity, both the trust network and the credit state must be distributed. We assume that each participating user runs an Ostra software agent on her own computer. This Ostra agent stores the user's key material and maintains secure network connections to the Ostra agents of the user's trusted friends. The two Ostra agents adjacent to a trust link each store a copy of the link's balance and bounds.

Ostra authorization requires a route computation in the trust network. Because user trust networks can be very large (many online social networks have hundreds of millions of users), the path computation must be scalable. Moreover, it is assumed that users wish to keep their trust relationships private. In a centralized design, such privacy can be ensured easily. In the decentralized design, this concern complicates the distributed route computation, as no user has a global view of the trust network.

In the sections below, we sketch out distributed designs for the route computation, for maintaining link balances and for ensuring that users follow the Ostra protocol.

7.2 Routing

Routing in large networks is a well-studied problem. We use a combination of existing techniques for distributed route discovery in large trust networks.

We divide the problem into two cases. To find routes within the local neighborhood of a user (e.g., all users within three hops), we use an efficient bloom filter-based [3] mechanism. To discover longer paths, we use landmark routing [23] to route to the destination's neighborhood and then use bloom filters to reach the destination. Each user creates and publishes a bloom filter (representing her local neighborhood) and a landmark coordinate (representing her location in the global network).

A user's bloom filter represents the set of users within the two-hop neighborhood of the user's trust network. Thus, given a destination's bloom filter, a user can determine whether any of her friends are within the destination's two-hop neighborhood. If so, the user has found the next hop toward the destination. The solution works on arbitrary connected graphs. However, the approach

is most efficient in sparse graphs in which the three-hop neighborhood accounts for a small percentage of the total network. Many real-world trust networks, such as social networks, have this property [18].

For long paths, we use landmark routing to reach the destination's neighborhood. A small subset of the user population is chosen as landmarks, and every user in the network determines her hop distance and the next hop to each of these landmarks. The landmarks are selected such that every user is within three hops of at least one landmark. Then, the resultant coordinate system can be used to route to within three hops of any destination user, and the bloom filters to reach the destination. Thus, given a destination user's coordinate, a user can first route to a landmark user who is "near" the destination, and this landmark user can then use bloom filter routing for the last few hops.

Preliminary analysis reveals that these two mechanisms are practical. On the YouTube social network from Section 6, more than 90% of users' bloom filters are smaller than 4 kilobytes. Additionally, with only 765 users (0.16% of the population) as landmarks, the remaining users can route to more than 89% of the network. The remaining, unrouteable users are mostly weakly connected and possess only a single link. In a real Ostra trust network, such users would seek more trust relationships, making them reachable as well.

7.3 Decentralized credit update

When the path in the trust network between the sender and receiver has been determined, the credit balances and bounds are updated in a decentralized manner during authorization, classification, and token expiration.

During authorization, the sender sends a signed authorization request message along the path. This request includes a unique identifier, the public key of the destination, and the destination's bloom filter and coordinate. Each user along the path (i) forwards the message, (ii) updates the balances and bounds of the message's incoming and outgoing links according to the rules stated below, (iii) records the destination, request identifier, previous hop, next hop, and expiration time of the request, and (iv) sets a timer for the expiration time. When the destination receives the request, it issues a signed token and sends it directly to the sender.

The link bounds are updated as follows. Each user along the path increments the lower bound L for the next hop, as was done in the centralized Ostra described in Section 4. Thus, the state of the network after a token is issued is exactly as shown in Figure 4 (b).

During classification, the destination sends a signed classification message along the path in the reverse direction. Each user checks if she has a record of a matching

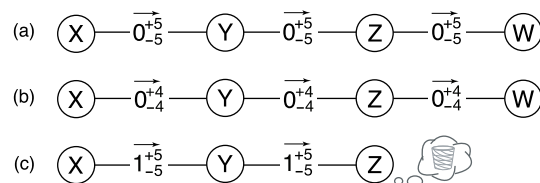


Figure 12: Diagram of how credit exchange occurs when X sends to W , with the penalty for dropping being one credit. The state of the link credits is shown (a) before the message is sent, (b) before the message is classified, and (c) after the timeout T if Z drops the message.

authorization request. If so, the adjustments of the link bounds performed during the authorization are undone, and the link balances are adjusted as described below. The message is then forwarded, and the record is deleted. Otherwise, if no matching record exists, the message is ignored.

The link balances are adjusted as was done in the centralized case. If the message was classified as wanted, the link balances are not changed, as shown in Figure 4 (d). However, if the message was classified as unwanted, each user raises the credit balance of the next hop in the path (the user to whom the original request was forwarded) and lowers the credit balance of the previous hop (the user from whom the original request was received). In this case, the resultant state of the network is shown in Figure 4 (c).

When the timer associated with an authorization request expires, then the user undoes the adjustments made to the link states during the authorization phase and deletes the request record.

Because authorization and classification messages are forwarded by the Ostra agents of users in the trust network, one concern is whether malicious users can simply drop such incoming messages. To protect against this, we provide users with an incentive to forward authorization requests and responses: users penalize the next hop along the path by lowering the next hop's credit if the message does not reach its destination.

Each user along the path adjusts the next hop's upper bound U by a penalty amount during the authorization phase. When the message is classified by the destination, the bound is restored. Otherwise, if a user drops the message, each of the users penalizes the next hop after the timeout T . An example is shown in Figure 12: while the message is pending classification (b), both the upper bound U and the lower bound L are changed to account for all possible outcomes. In the case in which Z drops the message (c), X penalizes Y , and Y penalizes Z . Thus, Z is penalized for dropping the message, whereas Y , who properly forwarded the message, has a neutral outcome.

8 Conclusion

We have described and evaluated Ostra, a system that leverages existing trust relationships among users to thwart unwanted communication. Unlike existing solutions, Ostra does not rely on strong user identities, does not depend on automatic content classification, and allows legitimate communication among users who have not had prior contact. Ostra can be applied readily to messaging and content-sharing systems that already maintain a social network. We have presented and evaluated a design of Ostra that works for systems with a trusted component (such as a Web site or a peer-to-peer system with a tracker). We have also sketched a design of Ostra which works with completely decentralized systems such as SMTP-based email. An evaluation based on data gathered from an online social networking site and an email trace shows that Ostra can effectively thwart unwanted communication while not impeding legitimate communication.

Acknowledgments

This work was supported in part by National Science Foundation grants ANI-0225660 and CNS-0509297. We would like to thank the anonymous reviewers and our shepherd Mike Dahlin for helpful comments, which greatly improved this paper. We would also like to thank Patrick Cernko for his assistance with the email trace used in the evaluation.

References

- [1] M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In *Proceedings of the Asian Computing Science Conference (ASIAN'03)*, Mumbai, India, December 2003.
- [2] S. Agarwal, V. N. Padmanabhan, and D. A. Joseph. Addressing email loss with SureMail: Measurement, design, and evaluation. In *Proceedings of the 2007 Usenix Annual Technical Conference (USENIX'07)*, Santa Clara, CA, June 2007.
- [3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [4] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System. In *Proceedings of the 5th ACM/USENIX Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.
- [5] A. Chapanond, M. S. Krishnamoorthy, and B. Yener. Graph Theoretic and Spectral Analysis of Enron Email Data. *Computational & Mathematical Organization Theory*, 11(3), October 2005.
- [6] J. Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS'02)*, Cambridge, MA, March 2002.
- [7] DSPAM. <http://dspam.nuclearelephant.com>.
- [8] R. Dunbar. Coevolution of neocortical size, group size and language in humans. *Behavioral and Brain Sciences*, 16(4):681–735, 1993.
- [9] Facebook. <http://www.facebook.com>.
- [10] S. E. Fahlman. Selling interrupt rights: A way to control unwanted e-mail and telephone calls. *IBM Systems Journal*, 41(4):759–766, 2002.
- [11] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. RE: Reliable Email. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, San Jose, CA, May 2006.
- [12] S. Hansell. Internet Is Losing Ground in Battle Against Spam. *The New York Times*, April 22, 2003.
- [13] L. G. Harbaugh. Spam-proof your in-box. *PCWorld*, May 2004.
- [14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'06)*, Pisa, Italy, August 2006.
- [15] J. R. Levine. An overview of e-postage. 2003. <http://www.taugh.com/epostage.pdf>.
- [16] LinkedIn. <http://www.linkedin.com>.
- [17] G. Mishne and D. Carmel. Blocking Blog Spam with Language Model Disagreement. In *Proceedings of the 1st International Workshop on Adversarial Information Retrieval on the Web (AIR-Web)*, Chiba, Japan, May 2005.
- [18] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/USENIX Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.
- [19] A. M. Odlyzko. The case against micropayments. In *Proceedings of Financial Cryptography: 7th International Conference*, Gosier, Guadeloupe, January 2003.
- [20] F.-R. Rideau. Stamps vs spam: Postage as a method to eliminate unsolicited commercial email. 2002. http://fare.tunes.org/articles/stamps_vs_spam.html.
- [21] J. Shetty and J. Adibi. The Enron Email Dataset Database Schema and Brief Statistical Report. Technical report, University of Southern California Information Sciences Institute, 2004.
- [22] SpamAssassin. <http://spamassassin.apache.org>.
- [23] P. F. Tsuchiya. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'88)*, Stanford, CA, August 1988.
- [24] M. Walfish, J. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed Quota Enforcement for Spam Control. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, San Jose, CA, May 2006.
- [25] YouTube. <http://www.youtube.com>.
- [26] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending against Sybil attacks via social networks. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'06)*, Pisa, Italy, August 2006.
- [27] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1994.

Detecting In-Flight Page Changes with Web Tripwires

Charles Reis
University of Washington

Steven D. Gribble
University of Washington

Tadayoshi Kohno
University of Washington

Nicholas C. Weaver
International Computer Science Institute

Abstract

While web pages sent over HTTP have no integrity guarantees, it is commonly assumed that such pages are not modified in transit. In this paper, we provide evidence of surprisingly widespread and diverse changes made to web pages between the server and client. Over 1% of web clients in our study received altered pages, and we show that these changes often have undesirable consequences for web publishers or end users. Such changes include popup blocking scripts inserted by client software, advertisements injected by ISPs, and even malicious code likely inserted by malware using ARP poisoning. Additionally, we find that changes introduced by client software can inadvertently cause harm, such as introducing cross-site scripting vulnerabilities into most pages a client visits. To help publishers understand and react appropriately to such changes, we introduce *web tripwires*—client-side JavaScript code that can detect most in-flight modifications to a web page. We discuss several web tripwire designs intended to provide basic integrity checks for web servers. We show that they are more flexible and less expensive than switching to HTTPS and do not require changes to current browsers.

1 Introduction

Most web pages are sent from servers to clients using HTTP. It is well-known that ISPs or other parties between the server and the client *could* modify this content in flight; however, the common assumption is that, barring a few types of client proxies, no such modifications take place. In this paper, we show that this assumption is false. Not only do a large number and variety of in-flight modifications occur to web pages, but they often result in significant problems for users or publishers or both.

We present the results of a measurement study to better understand what in-flight changes are made to web pages in practice, and the implications these changes

have for end users and web publishers. In the study, our web server recorded any changes made to the HTML code of our web page for visitors from over 50,000 unique IP addresses.

Changes to our page were seen by 1.3% of the client IP addresses in our sample, drawn from a population of technically oriented users. We observed many types of changes caused by agents with diverse incentives. For example, ISPs seek revenue by injecting ads, end users seek to filter annoyances like ads and popups, and malware authors seek to spread worms by injecting exploits.

Many of these changes are undesirable for publishers or users. At a minimum, the injection or removal of ads by ISPs or proxies can impact the revenue stream of a web publisher, annoy the end user, or potentially expose the end user to privacy violations. Worse, we find that several types of modifications introduce bugs or even vulnerabilities into many or all of the web pages a user visits—pages that might otherwise be safe and bug-free. We demonstrate the threats these modifications pose by building successful exploits of the vulnerabilities.

These discoveries reveal a diverse ecosystem of agents that modify web pages. Because many of these modifications have negative consequences, publishers may have incentives to detect or even prevent them from occurring. Detection can help publishers notify users that a page might not appear as intended, take action against those who make unwanted changes, debug problems due to modified pages, and potentially deter some types of changes. Preventing modifications may sometimes be important, but there may also be types of page changes worth allowing. For example, some enterprise proxies modify web pages to increase client security, such as Blue Coat WebFilter [9] and BrowserShield [30].

HTTPS offers a strong, but rigid and costly, solution for these issues. HTTPS encrypts web traffic to prevent in-flight modifications, though proxies that act as HTTPS endpoints may still alter pages without any indication to the server. Encryption can prevent even beneficial page

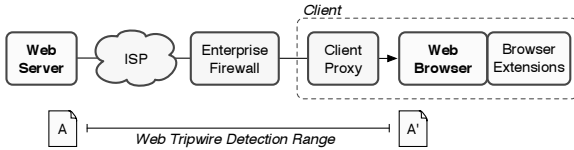


Figure 1: Web tripwires can detect any modifications to the HTML source code of a page made between the server and the browser.

changes, as well as web caching, compression, and other useful services that rely on the open nature of HTTP.

As a result, we propose that concerned web publishers adopt *web tripwires* on their pages to help understand and react to any changes made in flight. Web tripwires are client-side JavaScript code that can detect most modifications to unencrypted web pages. Web tripwires are not secure and cannot detect all changes, but they can be made robust in practice. We present several designs for web tripwires and show that they can be deployed at a lower cost than HTTPS, do not require changes to web browsers, and support various policy decisions for reacting to page modifications. They provide web servers with practical integrity checks against a variety of undesirable or dangerous modifications.

The rest of this paper is organized as follows. Section 2 describes our measurement study of in-flight page changes and discusses the implications of our findings. In Section 3, we compare several web tripwire implementation strategies that allow publishers to detect changes to their own pages. We evaluate the costs of web tripwires and their robustness to adversaries in Section 4. Section 5 illustrates how our web tripwire toolkit is easy to deploy and can support a variety of policies. Finally, we present related work in Section 6 and conclude in Section 7.

2 In-Flight Modifications

Despite the lack of integrity guarantees in HTTP, most web publishers and end users expect web pages to arrive at the client as the publisher intended. Using measurements of a large client population, we find that this is not the case. ISPs, enterprises, end users, and malware authors all have incentives to modify pages, and we find evidence that each of these parties does so in practice. These changes often have undesirable consequences for publishers or users, including injected advertisements, broken pages, and exploitable vulnerabilities. These results demonstrate the precariousness of today's web, and that it can be dangerous to ignore the absence of integrity protection for web content.



Figure 2: If a web tripwire detects a change, it displays a message to the user, as in the screenshot on the right.

To understand the scope of the problem, we designed a measurement study to test whether web pages arrive at the client unchanged. We developed a web page that could detect changes to its HTML source code made by an agent between the server and the browser, and we attracted a diverse set of clients to the page to test many paths through the network. Our study seeks to answer two key questions:

- What kinds of page modifications occur in practice, and how frequently?
- Do the changes have unforeseen consequences?

We found that clients at over 1% of 50,000 IP addresses saw some change to the page, many with negative consequences. In the rest of this section, we discuss our measurement technique and the diverse ecosystem of page modifications that we observed.

2.1 Measurement Infrastructure

Our measurement study identifies changes made to our web page between the web server and the client's browser, using code delivered by the server to the browser. This technique allows us to gather results from a large number of clients in diverse locations, although it may not detect agents that do not modify every page.

Technology. Our measurement tool consists of a web page with JavaScript code that detects page modifications. We refer to this code as a *web tripwire* because it can be unobtrusively placed on a web page and triggered if it detects a change. As shown in Figure 1, our web tripwire detects changes to HTML source code made anywhere between the server and browser, including those caused by ISPs, enterprise firewalls, and client-side proxies. We did not design the web tripwire to detect changes made by browser extensions, because extensions are effectively part of the browser, and we believe they are likely installed with the knowledge and consent of the user. In practice, browser extensions do not trigger the tripwire because they operate on the browser's inter-

nal representation of the page and not the HTML source code itself.

Our web tripwire is implemented as JavaScript code that runs when the page is loaded in the client's browser. It reports any detected changes to the server and displays a message to the user, as seen in Figure 2. Our implementation can display the difference between the actual and expected contents of the page, and it can collect additional feedback from the user about her environment. Further implementation details can be found in Section 3.

We note two caveats for this technique. First, it may have false negatives. Modifying agents may choose to only alter certain pages, excluding those with our web tripwires. We do not expect any false positives, though, so our results are a lower bound for the actual number of page modifications.¹ Second, our technique is not cryptographically secure. An adversarial agent could remove or tamper with our scripts to evade detection. For this study, we find it unlikely that such tampering would be widespread, and we discuss how to address adversarial agents in Section 4.2.

Realism. We sought to create a realistic setting for our measurement page, to increase the likelihood that agents might modify it. We included HTML tags from web authoring software, randomly generated text, and keywords with links.

We were also guided by initial reports of ISPs that injected advertisements into their clients' web traffic, using services from NebuAd [5]. These reports suggested that only pages from .com top-level domains (TLDs) were affected. To test this, our measurement page hosts several frames with identical web tripwires, each served from a different TLD. These frames are served from `vancouver.cs.washington.edu`, `uwsecurity.com`, `uwprivacy.org`, `uwsc.ca`, `uwsystems.net`, and `128.208.6.241`.

We introduced additional frames during the experiment, to determine if any agents were attempting to "whitelist" the domains we had selected to evade detection. After our measurement page started receiving large numbers of visitors, we added frames at `www.happyblimp.com` and `www2.happyblimp.com`.

In the end, we found that most changes were made indiscriminately, although some NebuAd injections were .com-specific and other NebuAd injections targeted particular TLDs with an unknown pattern.

Exposure. To get a representative view of in-flight page modifications, we sought visitors from as many vantage points as possible. Similar studies such as the ANA

Spoofers Project [8] attracted thousands of participants by posting to the Slashdot news web site, so we also pursued this approach.

Although our first submission to Slashdot was not successful, we were able to circulate a story among other sites via Dave Farber's "Interesting People" mailing list. This led another reader to successfully post the story to Slashdot.

Similarly, we attracted traffic from Digg, a user-driven news web site. We encouraged readers of our page to aid our experiment by voting for our story on Digg, promoting it within the site's collaborative filter. Within a day, our story reached the front page of Digg.

2.2 Results Overview

On July 24, 2007, our measurement tool went live at `http://vancouver.cs.washington.edu`, and it appeared on the front pages of Slashdot and Digg (among other technology news sites) the following day. The tool remains online, but our analysis covers data collected for the first 20 days, which encompasses the vast majority of the traffic we received.

We collected test results from clients at 50,171 unique IP addresses. 9,507 of these clients were referred from Slashdot, 21,333 were referred from Digg, and another 705 were referred from both Slashdot and Digg. These high numbers of referrals indicate that these sites were essential to our experiment's success.

The modifications we observed are summarized in Table 1. At a high level, clients at 657 IP addresses reported modifications to at least one of the frames on the page. About 70% of the modifications were caused by client-side proxies such as popup blockers, but 46 IP addresses did report changes that appeared to be intentionally caused by their ISP. We also discovered that the proxies used at 125 addresses left our page vulnerable to cross-site scripting attacks, while 3 addresses were affected by client-based malware.

2.3 Modification Diversity

We found a surprisingly diverse set of changes made to our measurement page. Importantly, these changes were often misaligned with the goals of the publisher or the end user. Publishers wish to deliver their content to users, possibly with a revenue stream from advertisements. Users wish to receive the content safely, with few annoyances. However, the parties in Figure 1, including ISPs, enterprises, users, and also malware authors, have incentives to modify web pages in transit. We found that these parties do modify pages in practice, often adversely impacting the user or publisher. We offer a high level survey of these changes and incentives below.

¹In principle, a false positive could occur if an adversary forges a web tripwire alarm. Since this was a short-term measurement study, we do not expect that we encountered any adversaries or false positives.

Category	IPs	ISP	Enterprise	User	Attacker	Examples
Popup Blocker	277			✓		Zone Alarm (210), CA Personal Firewall (17) , Sunbelt Popup Killer (12)
Ad Blocker	188			✓		Ad Muncher (99) , Privoxy (58), Proxomitron (25)
Problem in Transit	118	✓				Blank Page (107), Incomplete Page (7)
Compression	30	✓				bmi.js (23) , Newlines removed (6), Distillation (1)
Security or Privacy	17		✓	✓		Blue Coat (15), The Cloak (1) , AnchorFree (1)
Ad Injector	16	✓				MetroFi (6), FairEagle (5), LokBox (1), Front Porch (1), PerfTech (1), Edge Technologies (1), knects.net (1)
Meta Tag Changes	12		✓	✓		Removed meta tags (8), Reformatted meta tags (4)
Malware	3				✓	W32.Arpiframe (2), Adware.LinkMaker (1)
Miscellaneous	3			✓		New background color (1), Mark of the Web (1)

Table 1: Categories of observed page modifications, the number of client IP addresses affected by each, the likely parties responsible, and examples. Each example is followed by the number of IP addresses that reported it; examples listed in bold introduced defects or vulnerabilities into our page.

ISPs. ISPs have at least two incentives to modify web traffic: to generate revenue from advertising and to reduce traffic using compression. Injected advertisements have negative impact for many users, who view them as annoyances.

In our results, we discovered several distinct ISPs that appeared to insert ad-related scripts into our measurement page. Several companies offer to partner with ISPs by providing them appliances that inject such ads. For example, we saw 5 IP addresses that received injected code from NebuAd’s servers [2]. Traceroutes suggested that these occurred on ISPs including Red Moon, Mesa Networks, and XO, as well as an IP address belonging to NebuAd itself. Other frequently observed ad injections were caused by MetroFi, a company that provides free wireless networks in which all web pages are augmented with ads. We also observed single IP addresses affected by other similar companies, including LokBox, Front Porch, PerfTech, Edge Technologies, and knects.net.

Notably, these companies often claim to inject ads based on behavioral analysis, so that they are targeted to the pages a user has visited. Such ads may leak private information about a user’s browsing history to web servers the user visits. For example, a server could use a web tripwire to determine which specific ad has been injected for a given user. The choice of ad may reveal what types of pages the user has recently visited.

We also observed some ISPs that alter web pages to reduce network traffic. In particular, several cellular network providers removed extra whitespace or injected scripts related to image distillation [16]. Such modifi-

cations are useful on bandwidth-constrained networks, though they may also unintentionally cause page defects, as we describe in Section 2.4.1.

Enterprises. Enterprises have incentives to modify the pages requested by their clients as well, such as traffic reduction and client protection. Specifically, we observed proxy caches that remove certain meta tags from our measurement page, allowing it to be cached against our wishes. Such changes can go against a publisher’s desires or present stale data to a user. Our results also included several changes made by Blue Coat WebFilter [9], an enterprise proxy that detects malicious web traffic.

End Users. Users have several incentives for modifying the pages they receive, although these changes may not be in the best interests of the publishers. We found evidence that users block annoyances such as popups and ads, which may influence a publisher’s revenue stream. Users also occasionally modify pages for security, privacy, or performance.

The vast majority of page modifications overall are caused by user-installed software such as popup blockers and ad blockers. The most common modifications come from popup blocking software. Interestingly, this includes not only dedicated software like Sunbelt Popup Killer, but also many personal firewalls that modify web traffic to block popups. In both types of software, popups are blocked by JavaScript code injected into every page. This code interposes on calls to the browser’s `window.open` function, much like Naccio’s use of program rewriting for system call interposition [15].

Ad blocking proxies also proved to be quite popular. We did not expect to see this category in our results, be-

cause our measurement page contained no ads. That is, ad blocking proxies that solely removed ads from pages would have gone unnoticed. However, we detected numerous ad blocking proxies due to the JavaScript code they injected into our page. These proxies included Ad Muncher, Privoxy, Proxomitron, and many others.

Beyond these annoyance blocking proxies, we found user-initiated changes to increase security, privacy, and performance. AnchorFree Hotspot Shield claims to protect clients on wireless networks, and Internet Explorer adds a “Mark of the Web” comment to saved pages to prevent certain attacks [28]. Users also employed web-based anonymization services such as The Cloak [3], as well as proxies that allowed pages to be cached by removing certain meta tags.

Malware Authors. Surprisingly, our measurement tool was also able to detect certain kinds of malware and adware. Malware authors have clear incentives for modifying web pages, either as a technique for spreading exploit code or to receive revenue from injected advertisements. These changes are clearly adversarial to users.

In one instance, a client that was infected by Adware.LinkMaker [34] visited our measurement page. The software made extensive changes to the page, converting several words on the page into doubly underlined links. If the user hovered his mouse cursor over the links, an ad frame was displayed.

Two other clients saw injected content that appears consistent with the W32.Arpiframe worm [35]. In these cases, the clients themselves may not have been infected, as the Arpiframe worm attempts to spread through local networks using ARP cache poisoning [40]. When an infected client poisons the ARP cache of another client, it can then act as a man-in-the-middle on HTTP sessions. Recent reports suggest that web *servers* may also be targeted by this or similar worms, as in the recent case of a Chinese security web site [12].

2.4 Unanticipated Problems

In the cases discussed above, page modifications are made based on the incentives of some party. However, we discovered that many of these modifications actually had severe unintentional consequences for the user, either as broken page functionality or exploitable vulnerabilities. The threats posed by careless page modifications thus extend far beyond annoyances such as ad injections.

2.4.1 Page Defects

We observed two classes of bugs that were unintentionally introduced into web pages as a result of modifications. First, some injected scripts caused a JavaScript stack overflow in Internet Explorer when they

were combined with the scripts in our web tripwire. For example, the `xpopup.js` popup blocking script in CA Personal Firewall interfered with our calls to `document.write`. Similar problems occurred with a compression script called `bmi.js` injected by several ISPs. These bugs occasionally prevented our web tripwire from reporting results, but users provided enough feedback to uncover the issue. In general, such defects may occur when combining multiple scripts in the same namespace without the ability to sufficiently test them.

Second, we discovered that the CA Personal Firewall modifications interfered with the ability to post comments and blog entries on many web sites. Specifically, code injected by the firewall appeared in users’ comments, often to the chagrin of the users. We observed 28 instances of “`_popupControl()`” appearing on MySpace blogs and comments, and well over 20 sites running the Web Wiz Forums software [39] that had the same code in their comments. We reproduced the problem on Web Wiz Forums’ demo site, learning that CA Personal Firewall injected the popup blocking code into the frame in which the user entered his comments. We observed similar interference in the case of image distillation scripts that contained the keyword “nguncompressed.”

2.4.2 Vulnerabilities

More importantly, we discovered several types of page changes that left the modified pages vulnerable to cross-site scripting (XSS) attacks. The impact of these vulnerabilities should not be understated: the modifications made *most or all* of the pages a user visited exploitable. Such exploits could expose private information or otherwise hijack any page a user requests.

Ad Blocking Vulnerabilities. We observed exploitable vulnerabilities in three ad-blocking products: two free downloadable filter sets for Proxomitron (released under the names Sidki [33] and Grypen [19]), plus the commercial Ad Muncher product [4]. At the time of our study, each of these products injected the URL of each web page into the body of the page itself, as part of a comment. For example, Ad Muncher injected the following JavaScript comment onto Google’s home page:

```
// Original URL: http://www.google.com
```

These products did not escape any of the characters in the URL, so adversaries were able to inject script code into the page by convincing users to visit a URL similar to the following:

```
http://google.com/?</script><script>alert(1);
```

Servers often ignore unknown URL parameters (following the ‘?’), so the page was delivered as usual.

However, when Ad Muncher or Proxomitron copied this URL into the page, the “</script>” tag terminated the original comment, and the script code in the remainder of the URL was executed as part of the page. To exploit these vulnerabilities, an adversary must convince a user to follow a link of his own construction, possibly via email or by redirecting the user from another page.

It is worth noting that our measurement tool helped us discover these vulnerabilities. Specifically, we were able to search for page changes that placed the page’s URL in the body of the page. We flagged such cases for further security analysis.

We developed two exploit pages to demonstrate the threat posed by this attack. Our exploit pages first detect whether a vulnerable proxy is in use, by looking for characteristic modifications in their own source code (e.g., an “Ad Muncher” comment).

In one exploit, our page redirects to a major bank’s home page.² The bank’s page has a login form but is served over HTTP, not HTTPS. (The account name and password are intended to be sent over HTTPS when the user submits the form.) Our exploit injects script code into the bank’s page, causing the login form to instead send the user’s account name and password to an adversary’s server.

In a second exploit, we demonstrate that these vulnerabilities are disconcerting even on pages for which users do not normally expect an HTTPS connection. Here, our exploit page redirects to Google’s home page and injects code into the search form. If the user submits a query, further exploit code manipulates the search results, injecting exploit code into all outgoing links. This allows the exploit to retain control of all subsequent pages in the browser window, until the user either enters a new URL by hand or visits an unexploited bookmark.

In the case of Ad Muncher (prior to v4.71), any HTTP web site that was not mentioned on the program’s exclusion list is affected. This list prevents Ad Muncher from injecting code into a collection of JavaScript-heavy web pages, including most web mail sites. However, Ad Muncher did inject vulnerable code into the login pages for many banks, such as Washington Mutual, Chase, US Bank, and Wachovia, as well as the login pages for many social networking sites. For most social networking sites, it is common to only use HTTPS for sending the login credentials, and then revert to HTTP for pages within the site. Thus, if a user is already logged into such a site, an adversary can manipulate the user’s account by injecting code into a page on the site, without any interaction from the user. This type of attack can even be conducted in a hidden frame, to conceal it from the user.

²We actually ran the exploit against an accurate local replica of the bank’s home page, to avoid sending exploit code to the bank’s server.

In both Proxomitron filter sets (prior to September 8, 2007), all HTTP traffic is affected in the default configuration. Users are thus vulnerable to all of the above attack scenarios, as well as attacks on many web mail sites that revert to HTTP after logging in (e.g., Gmail, Yahoo Mail). Additionally, Proxomitron can be configured to also modify HTTPS traffic, intentionally acting as a “man in the middle.” If the user enables this feature, all SSL encrypted pages are vulnerable to script injection and thus leaks of critically private information.

We reported these vulnerabilities to the developers of Ad Muncher and the Proxomitron filter sets, who have released fixes for the vulnerabilities.

Internet Explorer Vulnerability. We identified a similar but less severe vulnerability in Internet Explorer. IE injects a “Mark of the Web” into pages that it saves to disk, consisting of an HTML comment with the page’s URL [28]. This comment is vulnerable to similar attacks as Ad Muncher and Proxomitron, but the injected scripts only run if the page is loaded from disk. In this context, the injected scripts have no access to cookies or the originating server, only the content on the page itself. This vulnerability was originally reported to Microsoft by David Vaartjes in 2006, but no fix is yet available [37].

The Cloak Vulnerabilities. Finally, we found that the “The Cloak” anonymization web site [3] contains two types of XSS vulnerabilities. The Cloak provides anonymity to its users by retrieving all pages on their behalf, concealing their identities from web servers. The Cloak processes and rewrites many HTML tags on each page to ensure no identifying information is leaked. It also provides users with options to rewrite or delete all JavaScript code on a page, to prevent the code from exposing their IP address.

We discovered that The Cloak replaced some tags with a comment explaining why the tag was removed. For example, our page contained a meta tag with the name “generatorversion.” The Cloak replaced this tag with the following HTML comment:

```
<!-- the-cloak note - deleting possibly dangerous
      META tag - unknown NAME 'generatorversion' -->
```

We found that a malicious page could inject script code into the page by including a carefully crafted meta tag, such as the following:

```
<meta name="foo--><script>alert(1);</script>">
```

This script code runs and bypasses The Cloak’s policies for rewriting or deleting JavaScript code. We reported this vulnerability to The Cloak, and it has been resolved as of October 8, 2007.

Additionally, The Cloak faces a more fundamental problem because it bypasses the browser’s “same origin

policy,” which prevents documents from different origins from accessing each other [31]. To a client’s browser, all pages appear to come from `the-cloak.com`, rather than their actual origins. Thus, the browser allows all pages to access each other’s contents. We verified that a malicious page could load sensitive web pages (even HTTPS encrypted pages) from other origins into a frame and then access their contents. This problem is already known to security professionals [20], though The Cloak has no plans to address it. Rather, users are encouraged to configure The Cloak to delete JavaScript code to be safe from attack.

OS Analogy. These vulnerabilities demonstrate the power wielded by web page rewriting software, and the dangers of any flaws in its use. An analogy between web browsers and operating systems helps to illustrate the severity of the problem. Most XSS vulnerabilities affect a single web site, just as a security vulnerability in a program might only affect that program’s operation. However, vulnerabilities in page rewriting software can pose a threat for *most or all* pages visited, just as a root exploit may affect all programs in an operating system. Page rewriting software must therefore be carefully scrutinized for security flaws before it can be trusted.

3 Web Tripwires

Our measurement study reveals that in-flight page modifications can have many negative consequences for both publishers and users. As a result, publishers have an incentive to seek integrity mechanisms for their content. There are numerous scenarios where detecting modifications to one’s own web page may be useful:

- Search engines could warn users of injected scripts that might alter search results.
- Banks could disable login forms if their front pages were modified.
- Web mail sites could debug errors caused by injected scripts.
- Social networking sites could inform users if they detect vulnerable proxies, which might put users’ accounts at risk.
- Sites with advertising could object to companies that add or replace ads.

Publishers may also wish to *prevent* some types of page changes, to prevent harm to their visitors or themselves.

HTTPS provides one rigid solution: preventing page modifications using encryption. However, the use of HTTPS excludes many beneficial services, such as caching by web proxies, image distillation by ISPs with

low bandwidth networks, and security checks by enterprise proxies. HTTPS also imposes a high cost on the server, in terms of financial expense for signed certificates, CPU overhead on the server, and additional latency for key exchange.

In cases where HTTPS is overly costly, we propose that publishers deploy web tripwires like those used in our measurement study. Web tripwires can effectively detect most HTML modifications, at low cost and in today’s web browsers. Additionally, they offer more flexibility than HTTPS for reacting to detected changes.

3.1 Goals

Here, we establish a set of goals a publisher may have for using a web tripwire as a page integrity mechanism. Note that some types of tripwires may be worthwhile even if they do not achieve all of the goals.

First, a web tripwire should detect any changes to the HTML of a web page after it leaves the server and before it arrives at the client’s browser. We exclude changes from browser extensions, as we consider these part of the user agent functionality of the browser. We also currently exclude changes to images and embedded objects, although these could be addressed in future work.

Second, publishers may wish for a web tripwire to prevent certain changes to the page. This goal is difficult to accomplish without cryptographic support, however, and it may not be a prerequisite for all publishers.

Third, a web tripwire should be able to pinpoint the modification for both the user and publisher, to help them understand its cause.

Fourth, a web tripwire should not interfere with the functionality or performance of the page that includes it. For example, it should preserve the page’s semantics, support incremental rendering of the page, and avoid interfering with the browser’s back button.

3.2 Designs & Implementations

Several implementation strategies are possible for building web tripwires. Unfortunately, limitations in popular browsers make tripwires more difficult to build than one might expect. Here, we describe and contrast five strategies for building JavaScript-based web tripwires.³ We also compare against the integrity properties of HTTPS as an alternative mechanism. The tradeoffs between these strategies are summarized in Table 2.

Each of our implementations takes the same basic approach. The web server delivers three elements to the browser: the *requested page*, a *tripwire script*, and a *known-good representation* of the requested page.

³We focus on JavaScript rather than Flash or other content types to ensure broad compatibility.

Goal	Count Scripts	Check DOM	XHR then Overwrite	XHR then Redirect	XHR on Self	HTTPS
Detects all HTML changes	✗	✓	✓	✓	✓	✓
Prevents changes*	✗	✗	✓	✗	✗	✓
Displays difference	✗	✗	✓	✓	✓	✗
Preserves semantics	✓	✓	✗	✓	✓	✓
Renders incrementally	✓	✓	✗	✗	✓	✓
Supports back button	✓	✓	✗	✗	✓	✓

Table 2: Comparison of how well each tripwire implementation achieves the stated goals. (*Neither “XHR then Overwrite” nor HTTPS can prevent all changes. The former allows full page substitutions; the latter allows changes by proxies that act as the encryption endpoint, at the user’s discretion.)

The known-good representation may take one of several forms; we use either a checksum of the page or a full copy of the page’s HTML, stored in an encoded string to deter others from altering it. A checksum may require less space, but it cannot easily pinpoint the location of any detected change. When all three of the above elements arrive in the user’s browser, the tripwire script compares the requested page with the known-good representation, detecting any in-flight changes.

We note that for all tripwire implementations, the web server must know the intended contents of the page to check. This requirement may sound trivial, but many web pages are simply the output of server-based programs, and their contents may not be known in advance. For these *dynamic* web pages, the server may need to cache the contents of the page (or enough information to reconstruct the content) in order to produce a tripwire with the known-good representation. Alternatively, servers with dynamic pages could use a web tripwire to test a separate static page in the background. This technique may miss carefully targeted page changes, but it would likely detect most of the agents we observed.

We have implemented each of the strategies described below and tested them in several modern browsers, including Firefox, Internet Explorer, Safari, Opera, and Konqueror. In many cases, browser compatibility limited the design choices we could pursue.

3.2.1 Count Scripts

Our simplest web tripwire merely counts the number of script tags on a page. Our measurement results indicate that such a tripwire would have detected 90% of the modifications, though it would miss any changes that do not affect script tags (*e.g.*, those made by the W32.Arpfirame worm). Here, the known-good representation of the page is simply the expected number of script tags on the page. The tripwire script compares against the number of script tags reported by the Document Object Model (DOM) to determine if new tags were inserted.

If a change is detected, however, it is nontrivial to determine which of the scripts do not belong or prevent them from running. This approach does miss many types of modifications, but it is simple and does not interfere with the page.

3.2.2 Check DOM

For a more comprehensive integrity check, we built a web tripwire that compares the full page contents to a known-good representation. Unfortunately, JavaScript code cannot directly access the actual HTML string that the browser received. Scripts only have access to the browser’s internal DOM tree, through variables such as `document.documentElement.innerHTML`. This internal representation varies between browsers and often even between versions of the same browser. Thus, the server must pre-render the page in all possible browsers and versions in order to provide a known-good representation of the page for any client. This technique is thus generally impractical.

Additionally, the server cannot always accurately identify a client’s user agent, so it cannot know which representation to send. Instead, it must send all known page representations to each client. We send a list of checksums to minimize space overhead. The tripwire script verifies that the actual page’s checksum appears in the array. Because checksums are used, however, this strategy cannot pinpoint the location of a change.

3.2.3 XHR then Overwrite

Rather than checking the browser’s internal representation of the page, our third strategy fetches the user’s requested page from the server as data. We achieve this using an `XmlHttpRequest` (XHR), which allows scripts to fetch the contents of XML or other text-based documents, as long as the documents are hosted by the same server as the current page. This is an attractive technique for web tripwires for several reasons. First, the

tripwire script receives a full copy of the requested page as a string, allowing it to perform comparisons. Second, the request itself is indistinguishable from a typical web page request, so modifying agents will modify it as usual. Third, the response is unlikely to be modified by browser extensions, because extensions expect the response to contain XML data that should not be altered. As a result, the tripwire script can get an accurate view of any in-flight modifications to the page.

In our first XHR-based web tripwire, the server first sends the browser a small *boot page* that contains the tripwire script and a known-good representation of the requested page (as an encoded string). The tripwire script then fetches the requested page with an XHR. It compares the response with the known-good representation to detect any changes, and it then overwrites the contents of the boot page, using the browser's `document.write` function.

This strategy has the advantage that it could *prevent* many types of changes by always overwriting the boot page with the known-good representation, merely using the XHR as a test. However, adversaries could easily replace the boot page's contents, so this should not be viewed as a secure mechanism.

Unfortunately, the overwriting strategy has several drawbacks. First, it prevents the page from rendering incrementally, because the full page must be received and checked before it is rendered. Second, the use of `document.write` interferes with the back button in Firefox, though not in all browsers. Third, we discovered other significant bugs in the `document.write` function in major browsers, including Internet Explorer and Safari. This function has two modes of operation: it can append content to a page if it is called as the page is being rendered, or it can replace the entire contents of the page if called after the page's `onload` event fires. Many web sites successfully use the former mode, but our tripwire must use the latter mode because the call is made asynchronously. We discovered bugs in `document.write`'s latter mode that can cause subsequent XHRs and cookie accesses to fail in Safari, and that can cause Internet Explorer to hang if the resulting page requests an empty script file. As a result, this overwriting approach may only be useful in very limited scenarios.

However, our measurement tool in Section 2 was small and simple enough that these limitations were not a concern. In fact, we used this strategy in our study.

3.2.4 XHR then Redirect

We made a small variation to the above implementation to avoid the drawbacks of using `document.write`. As above, the tripwire script retrieves the originally requested page with an XHR and checks it. Rather than

overwriting the page, the script redirects the browser to the requested page. Because we mark the page as cacheable, the browser simply renders the copy that was cached by the XHR, rather than requesting a new copy from the server. However, this approach still prevents incremental rendering, and it loses the ability to prevent any changes to the page, because it cannot redirect to the known-good representation. It also consistently breaks the back button in all browsers.

3.2.5 XHR on Self

Our final implementation achieves all of our stated goals except change prevention. In this XHR-based approach, the server first delivers the requested page, rather than a small boot page. This allows the page to render incrementally. The requested page instructs the browser to fetch an external tripwire script, which contains an encoded string with the known-good representation of the page. The tripwire script then fetches another copy of the requested page with an XHR, to perform the integrity check. Because the page is marked as cacheable (at least for a short time), the browser returns it from its cache instead of contacting the server again.⁴

This strategy cannot easily prevent changes, especially injected scripts that might run before the tripwire script. However, it can detect most changes to the requested page's HTML and display the difference to the user. It also preserves the page's semantics, the ability to incrementally render the page, and the use of the back button. In this sense, we view this as the best of the implementations we present. We evaluate its performance and robustness to adversarial changes in Section 4.

3.2.6 HTTPS

Finally, we compare the integrity properties of HTTPS with those of the above web tripwire implementations. Notably, the goals of these mechanisms differ slightly. HTTPS is intended to provide confidentiality and integrity checks for the *client*, but it offers no indication to the server if these goals are not met (e.g., if a proxy acts as the encryption end point). Web tripwires are intended to provide integrity checks for the *server*, optionally notifying the client as well. Thus, HTTPS and web tripwires can be seen to be complementary in some ways.

As an integrity mechanism, HTTPS provides stronger security guarantees than web tripwires. It uses encryption to detect all changes to web content, including images and binary data. It prevents changes by simply rejecting any page that has been altered in transit. It also

⁴If the page were not cached, the browser would request it a second time from the server. In some cases, the second request may see a different modification than the first request.

preserves the page’s semantics and ability to incrementally render.

However, HTTPS supports fewer policy decisions than web tripwires, such as allowing certain beneficial modifications. It also incurs higher costs for the publisher, as we discuss in Section 4.

4 Evaluation

To evaluate the strengths and weaknesses of web tripwires for publishers who might deploy them, we ask three questions:

1. Are web tripwires affordable, relative to HTTP pages without tripwires?
2. How do the costs of web tripwires compare to the costs of HTTPS?
3. How robust are web tripwires against adversaries?

We answer these questions by quantifying the performance of pages with and without web tripwires and HTTPS, and by discussing how publishers can react to adversarial page modifications.

4.1 Web Tripwire Overhead

To compare the costs for using web tripwires or HTTPS as page integrity mechanisms, we measured the client-perceived latency and server throughput for four types of pages. As a baseline, we used a local replica of a major bank’s home page, served over HTTP. This is a realistic example of a page that might deploy a tripwire, complete with numerous embedded images, scripts, and stylesheets. We created two copies of this page with web tripwires, one of which was rigged to report a modification. In both cases, we used the “XHR on Self” tripwire design, which offers the best strategy for detecting and not preventing changes. We served a fourth copy of the page over HTTPS, without a web tripwire.

All of our experiments were performed on Emulab [41], using PCs with 3 GHz Xeon processors. We used an Apache 2 server on Fedora Core 6, without any hardware acceleration for SSL connections.

Latency. For each page, we measured client-perceived latency using small scripts embedded in the page. We measured the start latency (i.e., the time until the first script runs) to show the responsiveness of the page, and we measured the end latency (i.e., the time until the page’s onload event fires) to show how long the page takes to render fully. We also measured the number of bytes transferred to the client, using Wireshark [14]. Our tests were conducted with a Windows XP client running Firefox, using a simulated broadband link with 2 Mbps

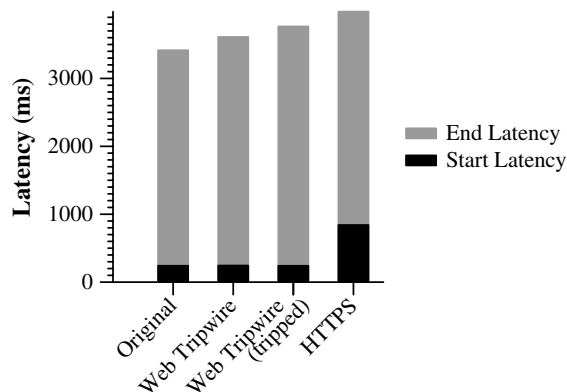


Figure 3: Impact of web tripwires and HTTPS on client perceived latency.

bandwidth and 50 ms one-way link latency. Each reported value is the average of 5 trials, and the maximum relative error was 3.25%.

Figure 3 shows that the pages with web tripwires did not increase the start latency over the original page (i.e., all were around 240 ms). In comparison, the extra round trip times for establishing an SSL connection contributed to a much later start for the HTTPS page, at 840 ms.

The time spent rendering for the web tripwires was longer than for the HTTP and HTTPS pages, because the tripwires required additional script computation in the browser. The web tripwire that reported a modification took the longest, because it computed the difference between the actual and expected page contents. Despite this, end-to-end latencies of the tripwire pages were still lower than for the HTTPS page.

Table 3 shows that transmitting the web tripwire increased the size of the transferred page by 17.3%, relative to the original page. This increase includes a full encoded copy of the page’s HTML, but it is a small percentage of the other objects embedded in the page.

Future web tripwire implementations could be extended to check all data transferred, rather than just the page’s HTML. The increase in bytes transferred is then proportional to the number of bytes being checked, plus the size of the tripwire code. If necessary, this overhead could be reduced by transmitting checksums or digests instead of full copies.

Throughput. We measured server throughput using two Fedora Core 6 clients running `htpferf`, on a 1 Gbps network with negligible latency. For each page, we increased the offered load on the server until the number of sustained sessions peaked. We found that the server was CPU bound in all cases. Each session simulated one visit to the bank’s home page, including 32 separate requests.

Technique	Data Transferred
Original	226.6 KB
Web Tripwire	265.8 KB
Web Tripwire (tripped)	266.0 KB
HTTPS	230.6 KB

Table 3: Number of kilobytes transferred from server to client for each type of page.

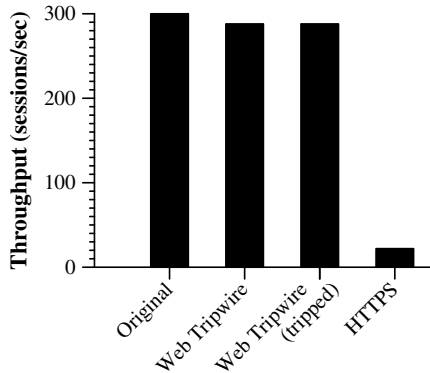


Figure 4: Impact of web tripwires and HTTPS on server throughput.

Figure 4 shows our results. The web tripwire caused only a 4% degradation of throughput compared to the original page. In comparison, the throughput dropped by over an order of magnitude when using HTTPS, due to the heavy CPU load for the SSL handshake.

For well-provisioned servers, HTTPS throughput may be improved by using a hardware accelerator. However, such hardware introduces new costs for publishers.

4.2 Handling Adversaries

In some cases, agents that modify web pages may wish for their behavior to remain undetected. For example, adversarial agents in the network may wish to inject ads, scripts, or even malicious code without being detected by the user or the publisher. Similarly, end users may wish to conceal the use of some proxies, such as ad-blockers, from the publisher.

In general, web tripwires cannot detect all changes to a page. For example, web tripwires cannot detect *full page substitutions*, in which an adversary replaces the requested content with content of his choice. Thus, we cannot address adversaries who are determined to deliver malicious content at all costs.

Instead, we consider a threat model in which adversaries wish to preserve the functionality of a page while introducing changes to it. This model assumes that ad-

versaries can observe, delay, and modify packets arbitrarily. However, it reflects the fact that end users often have some expectation of a page’s intended contents.

Under such a threat model, we hypothesize that publishers can make web tripwires effective against adversaries. Adversaries must both identify *and disable* any web tripwire on a page. Publishers can make both tasks difficult in practice using code obfuscation, using approaches popular in JavaScript malware for evading signature-based detection (e.g., code mutators [36], dynamic JavaScript obfuscation [43], and frequent code repacking [18]). Several additional techniques can challenge an adversary’s ability to identify or disable tripwires on-the-fly: creating many variants of web tripwire code, employing web tripwires that report an encoded value to the server even if no change is observed, and randomly varying the encoding of the known-good representation. Also, integrating web tripwire code with other JavaScript functionality on a page can disguise tripwires even if adversaries monitor the behavior of a page or attempt to interpret its code.

Ultimately, it is an open question whether an arms race will occur between publishers and agents that modify pages, and who would win such a race. We feel that the techniques above can help make web tripwires an effective integrity mechanism in practice, by making it more difficult for adversaries to disable them. However, using HTTPS (alternatively or in addition to web tripwires) may be appropriate in cases where page integrity is critical.

4.3 Summary

Overall, web tripwires offer an affordable solution for checking page integrity, in terms of latency and throughput, and they can be much less costly than HTTPS. Finally, though they cannot detect all changes, web tripwires can be robust against many types of agents that wish to avoid detection.

5 Configurable Toolkit

Based on our findings, we developed an open source toolkit to help publishers easily integrate web tripwires into their own pages. When using tripwires, publishers face several policy decisions for how to react to detected modifications. These include: (1) whether to notify the end user, (2) whether to notify the server, (3) whether the cause can be accurately identified, and (4) whether an action should be taken. Our toolkit is configurable to support these decisions.

The web tripwire in our toolkit uses the same “XHR on Self” technique that we evaluated in Section 4. We offer two implementations with different deployment sce-

narios: one to be hosted entirely on the publisher's server, and a second to be hosted by a centralized server for the use of many publishers.

The first implementation consists of two Perl CGI scripts to be hosted by the publisher. The first script produces a JavaScript tripwire with the known-good representation of a given web page, either offline (for infrequently updated pages) or on demand. The second script is invoked to log any detected changes and provide additional information about them to the user. Publishers can add a single line of JavaScript to a page to embed the web tripwire in it.

Our second implementation acts as a web tripwire service that we can host from our own web server. To use the service, web publishers include one line of JavaScript on their page that tells the client to fetch the tripwire script from our server. This request is made in the background, without affecting the page's rendering. Our server generates a known-good representation of the page by fetching a separate copy directly from the publisher's server, and it then sends the tripwire script to the client. Any detected changes are reported to our server, to be later passed on to the publisher. Such a web tripwire service could easily be added to existing web site management tools, such as Google Analytics [17].

In both cases, the web tripwire scripts can be configured for various policies as described below.

Notifying the User. If the web tripwire detects a change, the user can be notified by a message on the page. Our toolkit can display a yellow bar at the top of the page indicating that the page has changed, along with a link to view more information about the change. Such a message could be beneficial to the user, helping her to complain to her ISP about injected ads, remove adware from her machine, or upgrade vulnerable proxy software. However, such a message could also become annoying to users of proxy software, who may encounter frequent messages on many different web sites.

Notifying the Server. The web tripwire can report its test results to the server for further analysis. These results may be stored in log files for later analysis. For example, they may aid in debugging problems that visitors encounter, as proposed in AjaxScope [24]. Some users could construe such logging as an invasion of their privacy (e.g., if publishers objected to the use of ad blocking proxies). We view such logging as analogous to collecting other information about the client's environment, such as IP address and user agent, and use of such data is typically described under a publisher's privacy policy.

Identifying the Cause. Accurately identifying the cause of a change can be quite difficult in practice. It is clearly a desirable goal, to help guide both the user and pub-

lisher toward an appropriate course of action. In our own study, for example, we received feedback from disgruntled users who incorrectly assumed that a modification from their Zone Alarm firewall was caused by their ISP.

Unfortunately, the modifications made by any particular agent may be highly variable, which makes signature generation difficult. The signatures may either have high false negative rates, allowing undesirable modifications to disguise themselves as desirable modifications, or high false positive rates, pestering users with notifications even when they are simply using a popup blocker.

Our toolkit allows publishers to define patterns to match known modifications, so that the web tripwire can provide suggestions to the user about possible causes or decide when and when not to display messages. We recommend to err on the side of caution, showing multiple possible causes if necessary. As a starting point, we have built a small set of patterns based on some of the modifications we observed.

Taking Action. Even if the web tripwire can properly identify the cause of a modification, the appropriate action to take may depend highly on the situation. For example, users may choose to complain to ISPs that inject ads, while publishers may disable logins or other functionality if dangerous scripts are detected. To support this, our toolkit allows publishers to specify a callback function to invoke if a modification is detected.

6 Related Work

6.1 Client Measurements

Unlike web measurement studies that use a "crawler" to visit many servers, our work measures the paths from one server to many clients. Like the ANA Spoofer project [8], we drew many visitors by posting notices to sites like Slashdot and Digg. Opportunistic measurements of client traffic have been useful in other network studies as well, such as leveraging BitTorrent peers in iPlane [27], Coral cache users in Illuminati [10], and spurious traffic sources by Casado et al [11]. In particular, Illuminati also uses active code on web pages to measure properties of clients' network connections, and AjaxScope uses JavaScript to monitor web application code in clients' browsers [24].

6.2 Script Injection

We found that 90.1% of page modifications injected script code, showing that scripts play a prominent (but not exclusive) role in page rewriting. Interestingly, many publishers actively try to prevent script injection, as XSS attacks have had notable impact [1, 7].

Many such efforts aim to prevent attacks on the server, ranging from security gateways [32] to static analysis [42] or runtime protection [21]. These efforts do not prevent any injections that occur after a page leaves the server, so they do not address either the modifications or the vulnerabilities we discovered.

Some researchers defend against script injection on the client by limiting the damage that injected scripts can cause. These approaches include taint analysis [38] and proxies or firewalls that detect suspicious requests [22, 26]. Each of these approaches faces difficulties with false positives and false negatives, as they must infer unwanted behavior using heuristics.

BEEP proposes a whitelisting mechanism in which publishers can inform enhanced web browsers which scripts are authorized to run on a given web page [23]. The whitelist contains digests for each script fragment on the page, and the browser ignores any script fragment whose digest is not in the whitelist. Such whitelists can prevent browsers from running scripts injected in transit, as well as XSS attacks against vulnerable proxies like Ad Muncher and Proxomitron. However, whitelists would also prevent potentially desirable script injections, such as popup blockers, unless the browser granted exceptions for known scripts. BEEP's mechanism is no more secure than web tripwires, as it could also be modified in transit over an HTTP connection, and it cannot address modifications of other HTML tags than scripts.

6.3 Integrity Mechanisms

The name for our mechanism is inspired by the Tripwire project [25], an integrity checking mechanism for UNIX file systems. Tripwire detects changes to files by comparing their “fingerprints” to a known database. Our web tripwires achieve a similar goal for web pages, where the pages clients receive are analogous to the files Tripwire checks. In both cases, tripwires detect changes, notify administrators, and have configurable policies.

Methods for tamper-proofing software or content achieve similar goals, detecting unwanted changes to programs [13]. Also, the “XHR then Overwrite” strategy described in Section 3.2.3 has similarities to secure boot mechanisms such as AEGIS [6], both of which verify the integrity of an execution environment. In contrast, we forgo costly cryptographic mechanisms for inexpensive integrity tests, much like cyclic redundancy checks [29].

7 Conclusion

Using measurements of a large client population, we have shown that a nontrivial number of modifications occur to web pages on their journey from servers to

browsers. These changes often have negative consequences for publishers and users: agents may inject or remove ads, spread exploits, or introduce bugs into working pages. Worse, page rewriting software may introduce vulnerabilities into otherwise safe web sites, showing that such software must be carefully scrutinized to ensure the benefits outweigh the risks. Overall, page modifications can present a significant threat to publishers and users when pages are transferred over HTTP.

To counter this threat, we have presented “web tripwires” that can detect most modifications to web pages. Web tripwires work in current browsers and are more flexible and less costly than switching to HTTPS for all traffic. While they do not protect against all threats to page integrity, they can be effective for discovering even adversarial page changes. Our publisher-hosted and service-hosted implementations are easy to add to web pages, and they are available at the URL below:

<http://www.cs.washington.edu/research/security/webtripwires.html>

Acknowledgments

Hank Levy, Steve Balensiefer, and Roxana Geambasu provided useful feedback on drafts of this paper. We would also like to thank our shepherd, Paul Barham, and the anonymous reviewers for their suggestions. Scott Rose, Voradesh Yenbut, Erik Lundberg, and John Petersen helped prepare our servers for the “Slashdot effect,” and Ed Lazowska, Dave Farber, and Keith Dawson helped us gain wide exposure. We also thank the thousands of users who chose to visit our page to learn about changes to their web traffic.

This research was supported in part by the National Science Foundation under grants CNS-0132817, CNS-0430477, CNS-0627367, CNS-0722035, and NSF-0433702, by the Torode Family Endowed Career Development Professorship, and by gifts from Cisco and Intel.

References

- [1] Technical explanation of The MySpace Worm. <http://namb.la/popular/tech.html>, 2005.
- [2] NebuAd / Service Providers. <http://www.nebuad.com/providers/providers.php>, Aug. 2007.
- [3] The Cloak: Free Anonymous Web Surfing. <http://www.the-cloak.com>, Oct. 2007.
- [4] Ad Muncher. The Ultimate Popup and Advertising Blocker. <http://www.admuncher.com/>, 2007.
- [5] B. Anderson. fair eagle taking over the world? <http://benanderson.net/blog/weblog.php?id=D20070622>, June 2007.
- [6] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, May 1997.

- [7] C. Babcock. Yahoo Mail Worm May Be First Of Many As Ajax Proliferates. <http://www.informationweek.com/security/showArticle.jhtml?articleID=189400799>, June 2006.
- [8] R. Beverly. ANA Spoofer Project. <http://spoofer.csail.mit.edu/>, Nov. 2006.
- [9] Blue Coat. Blue Coat WebFilter. <http://www.bluecoat.com/products/webfilter>, Oct. 2007.
- [10] M. Casado and M. J. Freedman. Peering Through the Shroud: The Effect of Edge Opacity on IP-Based Client Identification. In *NSDI*, Apr. 2007.
- [11] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic Measurement: Extracting Insight from Spurious Traffic. In *HotNets-IV*, 2005.
- [12] Chinese Internet Security Response Team. ARP attack to CISRT.org. <http://www.cisrt.org/enblog/read.php?172>, Oct. 2007.
- [13] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation: Tools for Software Protection. In *IEEE Transactions on Software Engineering*, Aug. 2002.
- [14] G. Combs. Wireshark: The World's Most Popular Network Protocol Analyzer. <http://www.wireshark.org/>, Oct. 2007.
- [15] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.
- [16] A. Fox and E. A. Brewer. Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation. In *WWW*, May 1996.
- [17] Google. Google Analytics. <http://www.google.com/analytics/>, 2008.
- [18] D. Gryaznov. Keeping up with Nuwar. <http://www.avertlabs.com/research/blog/index.php/2007/08/15/keeping-up-with-nuwar/>, Aug. 2007.
- [19] Grypen. CastleCops: About Grypen's Filter Set. http://www.castlecops.com/t124920-About_Grypens_Filter_Set.html, June 2005.
- [20] B. Hoffman. The SPI laboratory: Jikto in the wild. <http://devsecurity.com/blogs/spilabs/archive/2007/04/02/Jikto-in-the-wild.aspx>, Apr. 2007.
- [21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *WWW*, May 2004.
- [22] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguichi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *AINA*, 2004.
- [23] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW*, May 2007.
- [24] E. Kiciman and B. Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *SOSP*, Nov. 2007.
- [25] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *CCS*, Nov. 1994.
- [26] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.
- [27] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *OSDI*, Nov. 2006.
- [28] Microsoft Developer Network. Mark of the Web. <http://msdn2.microsoft.com/en-us/library/ms537628.aspx>, Oct. 2007.
- [29] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. In *Proceedings of IRE*, Jan. 1961.
- [30] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *OSDI*, Nov. 2006.
- [31] J. Ruderman. The Same Origin Policy. <http://www.mozilla.org/projects/security/components/same-origin.html>, 2001.
- [32] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *WWW*, May 2002.
- [33] sidki. Proxomitron. <http://www.geocities.com/sidki3003/prox.html>, Sept. 2007.
- [34] Symantec.com. Adware.LinkMaker. http://symantec.com/security_response/writeup.jsp?docid=2005-030218-4635-99, Feb. 2007.
- [35] Symantec.com. W32.Arpiframe. http://symantec.com/security_response/writeup.jsp?docid=2007-061222-0609-99, June 2007.
- [36] P. Ször and P. Ferrie. Hunting For Metamorphic. In *Virus Bulletin Conference*, Sept. 2001.
- [37] D. Vaartjes. XSS via IE MOTW feature. <http://securityvulns.com/Rdocument866.html>, Aug. 2007.
- [38] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.
- [39] Web Wiz Guide. Web Wiz Forums - Free Bulletin Board System, Forum Software. <http://www.webwizguide.com/webwizforums/>, Oct. 2007.
- [40] S. Whalen. An Introduction to Arp Spoofing. http://www.rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf, Apr. 2001.
- [41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, Dec. 2002.
- [42] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, Aug. 2006.
- [43] B. Zdrnja. Raising the bar: dynamic JavaScript obfuscation. <http://isc.sans.org/diary.html?storyid=3219>, Aug. 2007.

Phalanx: Withstanding Multimillion-Node Botnets

Colin Dixon

Thomas Anderson
University of Washington

Arvind Krishnamurthy

Abstract

Large-scale distributed denial of service (DoS) attacks are an unfortunate everyday reality on the Internet. They are simple to execute and with the growing prevalence and size of botnets more effective than ever. Although much progress has been made in developing techniques to address DoS attacks, no existing solution is unilaterally deployable, works with the Internet model of open access and dynamic routes, and copes with the large numbers of attackers typical of today's botnets.

In this paper, we present a novel DoS prevention scheme to address these issues. Our goal is to define a system that could be deployed in the next few years to address the danger from present-day massive botnets. The system, called *Phalanx*, leverages the power of swarms to combat DoS. *Phalanx* makes only the modest assumption that the aggregate capacity of the swarm exceeds that of the botnet. A client communicating with a destination bounces its packets through a random sequence of end-host mailboxes; because an attacker doesn't know the sequence, they can disrupt at most only a fraction of the traffic, even for end-hosts with low bandwidth access links. We use PlanetLab to show that this approach can be both efficient and capable of withstanding attack. We further explore scalability with a simulator running experiments on top of measured Internet topologies.

1 Introduction

Botnets are very, very large. A recent estimate put the number of compromised machines participating in botnets at 150 million [6], while more modest estimates put the number around 6 million [3]. Single botnets regularly contain tens of thousands of end-hosts and have been seen as large as 1.5 million hosts [31]. These networks are the basis of a lucrative and difficult to detect underground economy on the Internet today. They steal identities and financial information, send most of the world's spam, and are used in DoS protection rackets [25]. These uses and less obvious ones have been extensively covered in previous work [32, 17]. To make matters worse, the number of critical operating system vulnerabilities discovered is increasing steadily every year [2] giving botnets an ample supply of new recruits, so the problem is unlikely to get better on its own.

Our focus in this paper is the impact of massive botnets on proposed solutions to denial of service (DoS) attacks. If compromised nodes are typical of end hosts

participating in other large peer-to-peer systems [18], a multimillion-node botnet would be able to generate over a terabit per second of traffic, sourced from virtually every routable IP prefix on the planet. This scale of attack could, at least temporarily, overwhelm any current core link or router! It is also capable of semi-permanently disrupting service to all but the best provisioned services on the Internet today. In May 2006, a sustained attack against the anti-spam company Blue Security forced the company to close down its services [20]. The damage is not just limited to security companies. Starting in April a sustained attack on government and business websites in Estonia effectively knocked the country off the web [16]. Even nations are not safe.

While the scale of the attacks may be increasing, the threat of DoS has been well understood for the past few years and many solutions have been proposed [23, 7, 9, 40, 19, 5, 39, 38, 33, 37, 30, 13, 35, 24, 26]. We believe that these solutions have not met with significant success in the real world because while often elegant, the burden of effective deployment is too high. The primary focus of our efforts is to provide a solution that can be effective when deployed unilaterally by as few parties as possible even if this means sacrificing some elegance.

It is worth noting that for read-only web sites, massive replication has proven an effective solution, and is even available as a commercial service [1]. However, the concern of this paper is non-replicable services, such as read/write back-end databases for e-commerce, modern AJAX applications, e-government, and multiplayer games, or point-to-point communication services such as VoIP or IM. For this case, we argue that no existing commercial or research solution is adequate to the scope of the problem posed by multimillion-node botnets.

In this paper, we propose a DoS prevention system, called *Phalanx*, that is capable of withstanding a multimillion-node botnet and yet can be reasonably deployed by a single large ISP today. *Phalanx* combines ideas from several ideas from prior work; co-designing the client, server, router support, and overlay software, to yield an effective yet remarkably simple system. Rather than being directly delivered, traffic is sent through a massive swarm of packet forwarders which act as mailboxes. Hosts use these mailboxes in a random order, so that even an attacker with a multimillion-node botnet can cause only a fraction of a given flow to be lost. Lightweight capabilities are then used to ensure that only

requested packets are delivered from the mailboxes to the destination.

Of course, to be practical, the system must not impose an undue burden in the absence of attack: we need to design the system to be cheap and unobtrusive. Ideally, the mechanisms will never be used—once attacks become ineffective, they will simply stop.

We expand on these ideas in more detail in the rest of the paper. Section 2 provides some relevant background and an overview of our approach. We present the complete architecture in Section 3, while Section 4 evaluates Phalanx’s performance and effectiveness. We discuss related work in Section 5, and conclude with Section 6.

2 Phalanx Overview

2.1 Background

While DoS attacks come in many flavors, we focus on resource exhaustion attacks. These attacks flood some bottleneck resource with more requests than can be handled ensuring that only a small fraction of the legitimate requests are serviced. The target resource is typically the weakest link on the path between the source and the destination, in terms of bandwidth, storage or computation.

Without information about which requests are legitimate and with limited buffer space, the only strategy for a victim is to serve requests at random. If there are G legitimate requests and B spurious requests, on average, $O(\frac{G}{B+G})$ of the available resources go to legitimate requests. B is often much larger than G , since with a massive botnet, attackers can pick their target and focus their fire. Addressing this asymmetry is a main goal of our work.

Of course, damage can be mitigated if traffic can be classified into legitimate and attack. We believe such classification is becoming increasingly difficult as botnets grow because attacks no longer need to spoof addresses or send abnormally large amounts of traffic per host in order to be successful. Clever attackers can simply emulate normal user behavior on each bot. While our approach is compatible with traffic classification, we explicitly do not rely upon it.

2.2 Assumptions

In approaching the general DoS problem, we make some assumptions about the network in which we operate.

- **Swarm:** We assume access to a large pool of well-provisioned machines which are geographically and topologically distributed. In essence, we assume access to a botnet of our own to absorb attacks. Our prototype is built and tested PlanetLab; as future work, we are exploring modifying a popular BitTorrent client to convert the millions of BitTorrent users into a community-based botnet defense.

- **Strong Adversary:** Botnets will continue to increase in size. We assume nodes in the botnet need not send anomalous amounts or kinds of traffic for attacks to be effective.
- **Minimal Network Support:** We assume that simple modifications to routers are feasible if they can be implemented in hardware at data rate at the edge of an ISP.
- **Predictable Paths:** Recent work [22] has shown that Internet paths can often be predicted without direct measurement of each possible path. This knowledge helps our system establish efficient yet resilient paths for good traffic.

2.3 Goals

A complete solution to the DoS problem must satisfy a large number of requirements including resistance to attack, deployability, performance, and backward compatibility. The following list of goals for our system helps to make these requirements concrete:

- **Eventual Communication:** Regardless of the number of attackers, it should be possible for a connection to eventually become established. In particular this means that a host needs to be able to perform a name lookup and acquire some form of capability to communicate even in the face of massive attack. Further, this must all be possible while keeping the current Internet’s open user model.
- **Protect Established Connections:** Once established, connections should be protected from collateral damage. Since any single path can easily be overwhelmed by a million-node botnet, each connection must leverage multiple paths to any destination. A connection should see degradation of at most $O(\frac{B}{B+G+M})$ where M is the set of mailboxes in Phalanx. In other words, performance should be proportional to the total number of good vs. bad nodes in the entire system, not the number of good vs. bad nodes focusing on a specific target.
- **Unilateral Deployment:** A single large ISP should be able to deploy an effective DoS solution for its customers, even from a massive attack, without needing to first reach a global agreement with all or most other ISPs.
- **Endpoint Policy:** The destination, and not the network, should control which connections are to be established and which packets are to be delivered.
- **Resistance to Compromise:** The system must tolerate compromised nodes and routers; its correct behavior should rely on only a few, simple (and thus possible to secure) functions.

- **Autoconfiguration:** Since Internet paths do change, both because of malicious activity and normal functioning, these changes should not interrupt protection.
- **Efficiency:** Communication performance should be close to that of the current Internet, even under attack. Otherwise an attack can be at least partially successful simply by evoking a response.

Many existing solutions achieve some of these goals, but as we explain later in Section 5, none achieves even the first three goals, much less all of them. We do leverage several key ideas from prior work, but we defer a detailed comparison until after we have described Phalanx.

3 Phalanx Architecture

In this section we describe the Phalanx architecture in detail. At a high level the architecture calls for sending all traffic through a set of mailboxes (Section 3.2) rather than directly to the destination. This approach requires some mechanism to prevent traffic from bypassing these mailboxes (Section 3.3) and also a system for handling connection setup (Section 3.4). Table 2.3 provides a summary of the mechanisms used in Phalanx along with their function, the goals they help achieve and where they are discussed in this paper. Additionally, the notation that we use to simplify discussion can be found in Table 3.2.1.

3.1 Components

Phalanx consists of three main components, which when combined, meet the goals laid out in Section 2.3. First, since it is easy for a large botnet to overwhelm any specific Internet path, we rely on a *swarm* which can match an attacking botnet in strength. This swarm puts legitimate users on the same footing as attackers by artificially inflating the number of “good” hosts for any given destination. The swarm appears to clients and servers as simple *packet mailboxes* using the API sketched in Table 3.2.1, allowing for a best effort packet store and pick-up point. These mailboxes are further explained in Section 3.2.

Second, a destination must explicitly request a packet from the mailboxes for it to be delivered; we use a set of Bloom filters at all of the border routers of the destination’s ISP to drop any unrequested packet (with high probability). This *filtering ring* implements an implicit per-packet network capability, rather than the per-connection capability [39, 40] or per-source address connectivity [10] in other proposals. This filtering ring is described in detail in Section 3.3.

Third, we use *resource proofs* and *authentication tokens* to facilitate connection setup. In an open Internet, there will often be no way to distinguish an initial connection request as being either good or from an attacker.

Resource proofs approximate fair queueing for these initial connection requests. We opt for computation-based proofs as in Portcullis [26] and OverDoSe [30] over bandwidth-based proofs [37]. The form of Phalanx resource proofs can be found in Section 3.4.3. Authentication tokens provide a way for clients with a preexisting relationship to a specific server to bypass its resource proof. Section 3.4.2 is a short description of Phalanx authentication tokens.

Together these components create a carefully chosen battleground where good users are on equal footing with attackers and can leverage the swarm’s resources to fight back against an attacking massive botnet.

3.2 Mailboxes

The basic building block in Phalanx is the packet mailbox. Mailboxes provide a simple abstraction that gives control to the destination instead of the source [7]. Rather than packets being delivered directly to the specified destination as in previous anti-DoS overlays [33, 19, 5], traffic is first delivered to a mailbox where it can either be “picked up” or ignored by the destination. Traffic which is ignored is eventually dropped from the buffers at packet mailboxes.

The interface which each mailbox exports is sketched in Table 3.2.1. The two basic operations are to *put* and *get* packets. The semantics are somewhat different from the traditional notions of *put* and *get*. A *put* inserts a packet into the mailbox’s buffer, possibly bumping an old entry, and returns. A *get* behaves somewhat less intuitively. Rather than behaving like a polling request, a *get* instead installs a best-effort interrupt at the mailbox. If a matching packet is found before the request is bumped from the buffer, the packet is returned. This is conceptually similar to *i3* [34], except triggers are valid for only one packet.

The mailbox abstraction puts the destination in complete control of which packets it receives. Flow policies can remain at the destination where the most information is available to make such decisions. These policies are implemented in the network via requests and the lack thereof. If no requests are sent, then no packets will come through. This behavior ensures that most errors are recoverable locally, rather than requiring cooperation and communication with the network control plane. This is in contrast to accidentally installing an overly permissive filter in the network and then being unable to correct the problem because the control channel can now be flooded.

3.2.1 Swarms & Iterated Hash Sequences

Mailboxes act as proxies, receiving and temporarily buffering traffic on behalf of end-hosts. If only a single such proxy existed, then we would just be moving the DoS problem from the end-host to the mailbox. In-

Mechanism	Function	Goals	Section
Mailboxes	lightweight network indirection primitive	Unilateral Deployment, Autoconfig	3.2
Iterated Hash Sequences	pseudo-random mailbox sequences	Protect Established Connections, Resistance to Compromise	3.2.1
Scalable Mailbox Sets	provide both efficiency and resilience according to current conditions	Efficiency	3.2.2
Filtering Ring	drop unrequested traffic before it can cause damage	Unilateral Deployment, Resistance to Compromise	3.3
General Purpose Nonces	allow small numbers of unrequested packets through the filtering ring	Eventual Communication	3.4.1
Cryptographic Puzzles	approximate fair queueing for connection establishment	Eventual Communication	3.4.3
Authentication Tokens	allow for pre-authentication of trusted flows	Eventual Communication	3.4.2
Congestion Control	adapt to access link heterogeneity	Autoconfig	3.5

Table 1: A summary of mechanisms used in Phalanx.

Symbol	Meaning
h	cryptographic hash function
x, y	shared secret to generate mailbox sequences
C	client, endpoint instigating the connection
S	server, endpoint receiving the connection
x_i	i th element of the sequence based on h and x
M	the set of mailboxes $\{M_i, \dots, M_{ M }\}$
$M[x_i]$	mailbox corresponding to x_i ($M_{x_i \bmod M }$)
K_Y	the public key belonging to Y
k_Y	the private key belonging to Y
$(z)_{K_Y}$	z encrypted using Y 's public key
$(z)_{k_Y}$	z signed using Y 's private key
w	the window size for requesting packets

Table 2: Notation

stead, we rely upon swarms of mailboxes to provide an end-host with many points of presence throughout the network. Assuming that the mailboxes' resources exceed that of attackers, legitimate clients will have some functioning channels for communication despite a widespread attack.

Individual flows are multiplexed over many mailboxes. Each packet in a flow is sent to a cryptographically random mailbox. Any given mailbox failure will only slightly affect a flow by causing a small fraction of the packets to be lost (often only a single packet). Since each mailbox is secretly selected by the endpoints, an attacker cannot "follow" a flow by attacking each mailbox just before it is used.

We construct a pseudo-random sequence of mailboxes during connection setup. The set of mailboxes M to use for this connection is determined by the destination, as described in the next section. The *sequence* of mailboxes is built by iterating a cryptographic hash function such as SHA-1 on a shared secret. We discuss how this secret is established in Section 3.4. Equipped with this shared sequence, both endpoints know in advance the precise mailbox to use for each packet in the connection.

putPacket(packet p)
places a packet in the local packet buffer
getPacket(nonce n)
places a request in the local packet buffer; when/if a packet arrives or has arrived it is returned
requestConnection(serverKey K_S)
asks for a challenge to earn access to establish a connection; returns a random nonce
submitSolution(string a , integer b)
provides a cryptographic puzzle solution to the challenge as a resource proof
submitToken(authentication token t)
provides proof of pre-authentication
issueNonces(signed nonce list N)
registers a set of general purpose nonces to be used for initial packet contact with the signing destination

Table 3: The Mailbox API.

To construct a sequence of mailboxes, we first define a sequence of nonces x_i based on the shared secret x and the cryptographic hash function h as follows.

$$x_0 = h(x||x)$$

$$x_i = h(x_{i-1}||x)$$

Including x in every iteration prevents an attacker who sniffs one nonce from being able to calculate all future nonces by iterating the hash function themselves. Our current implementation uses MD5 [29] as the implementation of h and thus uses 16-byte nonces for simplicity. This sequence of nonces then determines a corresponding sequence of mailboxes $M[x_i]$ by modulo reducing the nonces as follows.

$$M[x_i] = M_{x_i \bmod |M|}$$

Note that M need not be all mailboxes in the Phalanx deployment, as each flow can use a subset of the mailboxes. Indeed, a different set of mailboxes can be used

for each half of the flow (client-to-server and server-to-client); both sets can be dynamically re-negotiated within a flow.

One such shared secret and iterated hash function is used for each direction of communication. For the sake of discussion, we assume that the shared secret x generates the sequence x_i used for the client to server direction while the shared secret y generates the sequence y_i used for the server to client direction.

Each nonce serves as a unique identifier for a packet and is included in the header to facilitate pairing each incoming packet with its corresponding request. Thus the receiver can know precisely which source sent which packet. Including a nonce in each packet simplifies the logic needed to drop unrequested packets as described in Section 3.3. Lastly, nonces provide a limited form of authentication to requests; the attacker must snoop the nonce off the wire, and then deliver a replacement packet to a mailbox before the correct packet arrives, in order to subvert the system.

Communication proceeds with each packet and corresponding request going to the next mailbox in the sequence. Note that the data request is asynchronous with the data arrival—it may precede it or follow it at the mailbox (unlike i3 [34]). In the sequence given below, the requests precede the data packets, but in practice they will be sent simultaneously and the ordering does not matter.

$$\begin{aligned} M[x_i] &\leftarrow S: \text{request for } x_i \\ C \rightarrow M[x_i] &\rightarrow S: x_i, \text{ data} \\ M[x_{i+1}] &\leftarrow S: \text{request for } x_{i+1} \\ C \rightarrow M[x_{i+1}] &\rightarrow S: x_{i+1}, \text{ data} \end{aligned}$$

Here we only show one direction of communication, from the client to the server. The reverse direction proceeds in exactly the same way but selects mailboxes using the iterated hash sequence y_i based on the different shared secret y .

For flow control, each endpoint maintains a sliding window of w requested packets. This window is advanced each time a packet is received, or at the most recent packet rate if all packets in the window are lost. For optimal throughput, the window should be at least as large as the bandwidth-delay product, but there are several advantages to keeping w modest in size. First, w represents the number of outstanding requests which might be received all at once from a malicious sender. A wily attacker may delay delivering packets to mailboxes until as many gets have been registered as possible.

Second, keeping w small reduces the length of time packets are queued at mailboxes before being requested (or equivalently requests are queued before being matched). Recall that mailbox queueing is best effort

and thus packets and requests can be dropped. We assume that each mailbox is well-provisioned in that it can queue at least a few seconds of packets and requests in DRAM at its network access link bandwidth.

3.2.2 Mailbox Sets

Picking the subset of mailboxes in Phalanx to use for a specific connection poses a tradeoff of performance and resilience. For best performance, we would like mailboxes that are only slight detours off the best path to the destination; for resilience, we would like mailboxes that exhibit the greatest path diversity to the filtering ring. We opt to make the mailbox sets dynamically negotiable within each flow to get the best of both worlds. Initially, we start each connection with a small number of mailboxes (ten in the prototype), chosen to achieve good resilience without sacrificing performance. In the prototype, we use iPlane's route and performance predictions [22] to guide this choice. If the flow sees significant loss, indicating a possible attack, additional mailboxes are added to increase path diversity.

For example, a connection from Los Angeles to Seattle might start off using mailboxes in different ISPs in San Francisco and Portland for their low additional latency. If loss rates cross a given threshold, then mailboxes in Denver and Salt Lake City can be added to increase path diversity. If the attack continues and becomes more severe, the connection might start redirecting packets through mailboxes across the US, Asia and Europe. If an attack is ongoing, new connections might be started off with a larger and more diverse set of initial mailboxes. As an optimization which we do not yet implement, mailbox sets can be passed by reference. The set of all mailboxes for a particular destination is relatively static and can be widely distributed; a particular connection need only agree on which random subset of this list to use, e.g., by hashing on the shared connection secret.

Because both endpoints need to be using the same set of mailboxes to ensure proper delivery, re-negotiating the set of mailboxes in the middle of a connection is not trivial. By default, Phalanx uses two mailbox sets (one for each direction) and each endpoint controls the set which it receives through. If an endpoint wishes to make any changes, it piggybacks (a pointer to) the new set in a normal packet, along with a first sequence number for which the changes will be valid.

The endpoint then waits for that sequence number and branches in both directions: one assuming the changes were received successfully, the other assuming that they were lost. Whenever one branch receives a packet, the endpoint knows which branch was correct and drops the other branch. If the change request arrives at the sender too late (i.e., after the packet with the sequence number has already been sent on the old path), the sender must

ignore it; the receiver is then free to try again. This provides support for dynamic re-negotiation and thus helps provide both performance and resiliency according to the prevailing conditions.

3.3 Filtering ring

With Phalanx, a protected destination only receives those packets which it explicitly requests from a mailbox. To enforce this, we drop all other packets for the destination at the edge of its upstream ISP (See Figure 1). This means that a protected destination cannot serve as a mailbox, and more importantly, this breaks legacy clients. We have not implemented legacy client support in our prototype yet, but we envision an applet that a web site would provide its clients to mediate their access through Phalanx. This does not create a “chicken and egg” problem, because the applet would be read-only data that can be widely replicated; as we observed earlier, existing commercial anti-DoS solutions are effective for distributing read-only data.

Implementing the filtering ring is straightforward, even at hundred gigabit data rates. Each request packet carries a unique nonce that allows a single reply packet to return. In the simple case of symmetric routes, the border router records the nonce on the outgoing packet, and matches each incoming packet to a recently stored nonce, discarding the packet if there is no match. Each nonce is single use, so that once an incoming packet has matched a nonce, we remove that nonce.

Of course, an attacker might try to flood the border router (or more precisely, the links immediately upstream from the border router) to prevent returning packets from ever reaching the destination. As we observed earlier, a massive botnet may be able to flood any single link or router in the network. However, this would disconnect only those mailboxes that used that specific router to access the destination; other mailboxes would continue to deliver packets unaffected. Even a multimillion-node botnet would be unable to sustain enough traffic to completely disconnect a tier-1 ISP from the Internet. (To have an effective defense against such a large scale attack, a destination must either be a direct customer of a tier-1 that provides a filtering ring, or be protected indirectly, as customer of an ISP that is a customer of that tier-1.) Since each connection can spread its packets across a diverse set of mailboxes, connections might experience a higher packet loss rate during an attack, but otherwise would continue to make progress.

Our implementation of the filtering logic uses two lists of nonces, efficiently encoded using Bloom filters [12]. A whitelist contains a list of requested nonces while a blacklist contains a list of nonces which have already entered the filtering ring. The whitelist ensures that only requested packets get through, while the blacklist ensures

that at most one packet gets through per request. As request packets leave the ring, the router adds their nonces to the local whitelist. When data packets enter the ring, their nonces are verified by checking the whitelist and then added to a blacklist. Bloom filters must be periodically flushed to work properly; to minimize the impact of these flushes, two copies of each list are maintained and they are alternately flushed.

While Bloom filters provide only probabilistic guarantees, this is sufficient for our purposes as they do not yield false negatives. Even in the unlikely event that an attacker’s guessed nonce is a false positive for the whitelist it will then be added to the blacklist making it good for only one packet. It is not possible that a correct returning packet will miss in the whitelist because that would require a false negative. There is still a (small) possibility that a legitimate packet will be incorrectly dropped because of a collision in the blacklist, but Phalanx is designed to be robust to packet loss.

We believe that the Phalanx filtering ring is efficient enough to be implemented even for high speed links inside the core of the Internet, provided there is an incentive for ISPs to deploy the hardware, that is, provided that ISPs can charge their customers for DoS protection. (Note that ISPs that provide transit need to modify only their ingress routers and not all routers.) A 100 gigabit router line card would need about 50MB of hash table space. For each delivered packet, six Bloom filter operations are required: the request packet places a nonce in the current copy of the whitelist, then when the actual packet is received it is checked against all four tables (the current and previous whitelist, and the current and previous blacklist), and then added to the current blacklist. Both the storage and computation are small relative to those needed for core Internet routing tables and packet buffering.

To be effective, the filtering ring must be comprehensive – able to examine every packet destined for a protected destination, regardless of the source of the traffic. Bots are everywhere, even inside corporate networks. As a result, it seems likely that filtering rings would be deployed in depth, as shown in Figure 1. Initially, small scale ISPs close to the destination could offer a limited DoS protection service, capable of withstanding moderate-sized botnets. Moving outward, the cost of deploying the filtering ring would increase (more border routers to upgrade), but the value would also increase as the system would be able to withstand larger-scale botnets.

Our discussion to this point has assumed routing symmetry. Of course, the real Internet has a substantial amount of routing asymmetry. A request packet sent to a mailbox may not leave the filtering ring at the same point as the corresponding data packet returns; if so, the Bloom

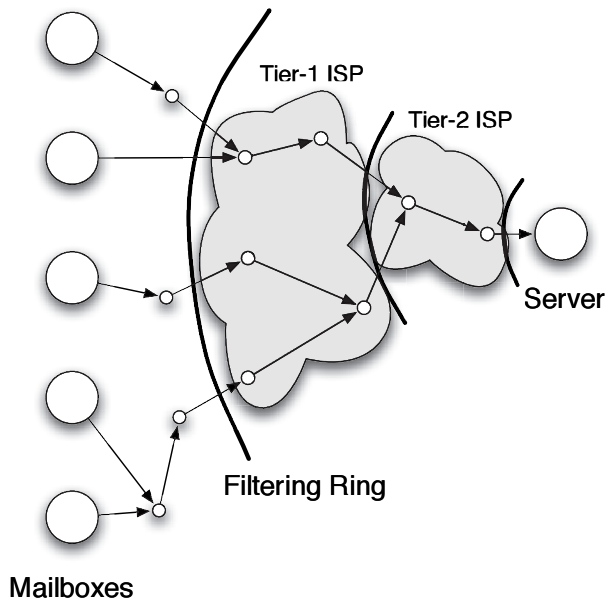


Figure 1: Filtering ring placement. Unlike in current Internet routing where all packets for a single connection take a single path across each ISP boundary, in Phalanx packets from a given connection may use many different paths across the set of filtering rings.

filter at the return point will drop the packet. This problem becomes more likely as the nesting level increases.

To address this problem, we allow destinations to loosely source route request packets via IP-in-IP tunneling to each mailbox. The source route is chosen to be the reverse of the predicted sequence of filtering ring nodes from the mailbox back to the destination. This guarantees that each filtering ring will be appropriately primed. We use iPlane to gather the data to make the route prediction [22].

3.4 Connection Establishment

Thus far, we have described a way for a destination to receive packets it is expecting. In order to establish a connection, a first (unexpected) packet must be delivered.

We allow for connection establishment by issuing periodic requests which ask for connection establishment packets rather than specific data packets. These *general purpose nonces* are described in Section 3.4.1. Simply allowing for such first packets doesn't solve the problem as they immediately become a scarce resource and this capability acquisition channel can be attacked [8]. To solve this problem, we require clients to meet some burden before giving them access to a general purpose nonce. Clients can either present an *authentication token* signed by the server as explained in Section 3.4.2 or present a *cryptographic puzzle* solution as explained in Section 3.4.3.

3.4.1 Passing through the filtering ring

Rather than invent new mechanisms to deal with allowing first packets through the filtering ring, we reuse the existing request packet framework to punch nonspecific holes in the filtering ring. Destinations send each mailbox a certain rate of general purpose requests. Each request contains a nonce to be placed in such first packets. When a mailbox wishes to send a first packet, it places one of these general purpose nonces into the packet allowing it to pass through the filtering ring.

These general purpose requests implement a form of admission control. Each general purpose nonce announces the destination's willingness to admit another flow. This further increases the destination's control over the traffic it receives, allowing it to directly control the rate of new connection requests.

In order for the general purpose nonce mechanism to be resilient to DoS attack, it is necessary to spread them across a wide set of well-provisioned mailboxes; a particular client only needs to access one. Refreshing these general purpose nonces can pose an unreasonable overhead for destinations that receive few connection requests, and as a result, our prototype supports nonces issued for aggregates of IP addresses. Thus, an ISP can manage general purpose nonces on behalf of an aggregate of users, at some loss in control over the rate of new connections being made to each address. Of course, the ISP must carefully assign aggregates based on their capacity to handle new connection requests; for example, google should not be placed in the same aggregate as a small web site, or else the attacker could use general purpose nonces to flood the small site. An ISP already manages the statistical multiplexing of resources We discuss in Section 3.6 how the client learns the initial set of mailboxes to use for a specific destination.

When a client wishes to contact some server, it first contacts a mailbox and asks that mailbox to insert a general purpose nonce into its first packet and forward it to the destination. Because general purpose nonces are a scarce resource, the mailbox needs rules governing which connections to give these nonces and in what order. The next two sections deal with those mechanisms.

3.4.2 Authentication tokens

Each packet requesting to initiate a connection must either carry an authentication token or a solution to a cryptographic puzzle. These provide the burden of proof necessary for a mailbox to allow access to general purpose nonces. Authentication tokens provide support for pre-authenticated connections allowing them to begin with no delay; for example, a popular e-commerce site such as Amazon might provide a cookie to allow quicker access to its web site to its registered users. Cryptographic puzzles provide resource proofs to approximate fair queue-

ing of requests, when no prior relationship exists between source and destination.

Authentication tokens are simply a token signed by the server stating that the given client is allowed to contact that server. An additional message exchange is required to prove that the client is in fact the valid token holder. Authentication tokens take the form $(K_C, t)_{k_S}(N)_{k_C}$.

The first portion is the public key of the client and an expiration time signed by the server. This represents that the server has given the client the right to initiate connections until the listed expiration time. The second portion is a random fresh nonce issued by a mailbox and signed by the client. This proves that the client is in control of the private key to which the token was originally issued.

The authentication token connection establishment protocol then proceeds as follows:

```

C → M[*]: request challenge
C ← M[*]: N
C → M[*] → S: N, (K_C, t)_{k_S}(N + 1)_{k_C}, (K_C, x)_{k_S}
C → M[x_0]: request for x_0
C ← M[x_0] ← S: x_0, (K_S, y)_{K_C}, data
M[y_0] ← S: request for y_0
C → M[y_0] → S: y_0, data
C → M[x_1]: request for x_1
⋮

```

First a challenge nonce is requested and received. The nonce is created to be self certifying as in SYN cookies [11] by making it a secret cryptographic hash of the mailbox and client IP addresses and ports as well as some local, slowly-increasing timer. This prevents a SYN flood style attack on memory at the mailbox.

Next the authentication token is presented along with the client's public key and a shared secret to be forwarded to the server. If the token is valid, the mailbox forwards the whole request on to the server which can then choose to accept the connection or not as it sees fit.

3.4.3 Crypto-puzzles

The crypto-puzzle is designed to be a resource proof allowing hosts which spend more time solving the puzzle to get higher priority to the limited number of general purpose nonces each mailbox possesses. While there are many kinds of resource proofs, we opt for a computational resource proof rather than a bandwidth resource proof [37] because computation tends to be much cheaper and less disruptive when heavily used.

We borrow the solution presented in Portcullis [26] and OverDoSe [30] where the crypto-puzzle used is to find a partial second pre-image of a given random challenge string such that, when hashed, both strings match in the lower b bits. The goal for each client is then to find

some string a given a challenge nonce N such that:

$$h(a||N) \equiv h(N) \bmod 2^b$$

The random nonce is included in both strings to prevent attackers from building up tables of strings which cover all 2^b possible values of the lower b bits in advance. In effect, they need to cover $2^{b+|N|}$ possible values to find matches for all values of the lower b bits and for all possible nonces, whereas solving the puzzle online need only search 2^{b-1} strings on average. Because the length of the nonces is under the control of the mailboxes, it is possible to make the pre-computing attack arbitrarily harder than waiting and solving puzzles online.

First packets are granted general purpose nonces with priority given first to those with valid authentication tokens and then in decreasing order of matching bits in the crypto-puzzle solution. This allows any source to get a first packet through against an attacker using only finite resources per first packet albeit at an increase in latency.

Assuming that attackers have some fixed computational ability, we know that there is some number of bits b_a such that if attackers continuously solved cryptographic puzzles in b_a bits, they would not be able to consume all general purpose requests. Knowing this, it is easy to show that any client which solves a crypto puzzle in b_a bits will be guaranteed to get a general purpose nonce.

If the attackers solve puzzles in b_a or more bits, then by definition of b_a there will be left over general purpose nonces for the client to use. If the attackers solve puzzles in fewer than b_a bits, then the client will preferentially receive general purpose nonces based on the priority queuing.

Knowing b_a in advance is not necessary because solving puzzles in all numbers of bits from 1 to b_a only takes twice as long as solving a puzzle in b_a bits. Thus, at a cost of a factor of 2 in latency, we can try all possible numbers of bits until we find the "correct" number.

In addition to reducing overhead, aggregating general purpose nonces across multiple IP addresses has one further benefit. Without aggregation, a botnet can amass its resources to drive up the cost of acquiring all of the nonces for a specific destination. With aggregation, the botnet must compete with a larger number of good clients, and a faster refresh rate of general purpose nonces, in order to target any specific destination.

Connection establishment using crypto-puzzles proceeds as follows:

```

C → M[*]: request challenge
C ← M[*]: N
C → M[*] → S: N, a, h(a||N), b, (KC, x)KS
C → M[x0]: request for x0
C ← M[x0] ← S: x0, (KS, y)KC, data
M[y0] ← S: request for y0
C → M[y0] → S: y0, data
C → M[x1]: request for x1
⋮

```

With the exception of the substitution of a crypto-puzzle solution for an authentication token, this is identical to the solution presented in Section 3.4.2.

3.5 Congestion Control

Forwarding traffic through mailboxes with significant path diversity makes Phalanx interact somewhat poorly with normal TCP. Packets will be frequently reordered, and losses from one mailbox should not necessarily cause a connection to reduce its rate. Although this scenario bears resemblance to multipath congestion control, the issue in Phalanx is further complicated by the fact that attackers can exploit congestion response as an avenue for DoS [21, 27].

Instead, we build a simple congestion control protocol based on the assumption that congestion only occurs at the access links of the sender, receiver and/or mailbox. Receivers advertise a maximum packet rate they are willing to receive from a particular sender, based on local policies and available resources. A sender uses the receiver's advertised rate along with current observed packet receipt rate to adjust its sending rate.

Essentially, the receiver picks a rate it is willing to receive, say 50 packets per second. To begin with the sender sends 50 packets a second through its set of mailboxes. If losses bring the rate below 50 packets a second, indicating either congestion or a DoS attack, the sender will ramp *up* its packet rate to compensate, using either forward error correction or retransmissions to recover lost packets. While this is not TCP friendly, it is impossible to be both TCP friendly and resistant to flooding attacks.

It is also possible for mailboxes to become overloaded, e.g., due to true congestion. A mailbox experiencing congestion is free to simply drop packets, but it can also set the IP ECN bit in packets passing through it. This signals to the destination that it should reduce the rate through this particular mailbox. The destination can achieve this by simply removing that mailbox from future flows, or reconfiguring the mailbox sets of existing flows to exclude the congested mailbox. In our prototype, we have implemented a finer-grained approach. Each mailbox in the mailbox set is assigned a weight,

corresponding to the estimate of how much traffic that mailbox can successfully handle. This weight is used to bias the random selection of mailboxes to favor those that can handle more traffic; the weights can be dynamically adjusted, in the same manner as in Section 3.2.2, by agreement between the source and destination.

3.6 Name Service Lookup

In order to provide a complete DoS solution, all components of a connection must be protected, from looking up the server, to logging in, to closing the connection. As we attempt to solve the more general problem of providing DoS protection for typical public server on the current Internet, protecting lookup is as important as protecting the actual connection.

Fortunately, lookup services typically serve small amounts of static content and can thus be highly replicated to provide DoS resilience. Other highly replicated name services based on DHT like CoDoNS [28] provide such solutions. One approach would be to run the DHT-based lookup service on top of all mailboxes.

Rather than returning the address of a server, the name service instead returns a list of mailboxes willing to handle connections for the server. As before, this list of mailboxes can be chosen using iPlane in the absence of an attack, or taken randomly from a topologically diverse list to provide better resilience. These points of contact can differ from the mailboxes which will eventually be used for normal communication; those mailboxes are negotiated during connection setup.

The server must be able to update these records, but we imagine the set of first contact mailboxes will not change especially quickly and it is not necessary for the change to be atomic. This enables almost any off-the-shelf DHT semantics to serve our purposes.

3.7 A Working Example

To illustrate the complete system in action, we consider the example of fetching a web page from a Phalanx-protected server. The example can also be followed in Figure 2 where the numbered steps will be mentioned.

First, the client looks up the address of the server for and subsequently requests the static, cacheable content of the page via any current CDN-style system with high-availability, such as Akamai, CoDeeN or Coral (as in steps 1 and 2). As part of fetching this content, the client receives a static and cacheable Java applet, which then serves as a zero-installation client to allow for interaction with Phalanx mailboxes. At this point, the Java applet is responsible for rendering the dynamic, non-cacheable portions of the page and speaking the Phalanx protocols.

The applet begins by making a name request for the dynamic content server to the distributed name service (3). Again, because the naming information is static

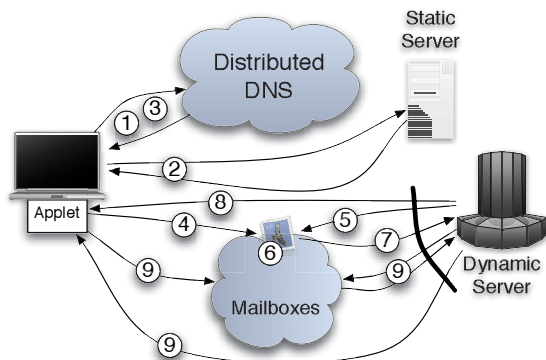


Figure 2: A diagram illustrating a simple HTTP-style request done with Phalanx. The numbers correspond to the description in Section 3.7.

and cacheable, this service can be provided by any highly available name service, such as CoDoNs or Akamai's DNS service. The name service returns a list of "first-contact" mailboxes. These first-contact mailboxes hold general purpose nonces that the server has issued to allow new connections to be made.

The applet requests a challenge nonce from one of these mailboxes and replies with either a puzzle solution or an authentication token (4). In either case, the applet waits some period of time for a response before retrying the request possibly with a more complex puzzle solution and/or trying a different mailbox.

At the mailbox, a steady stream of general purpose nonces has been arriving from the dynamic content server (5). One of these general purpose nonces is eventually assigned to the client's connection request (6) at which point the applet's request is forwarded to the server along with the general purpose nonce (7) to cross back through the filtering rings without being dropped.

Eventually, a response will come back from the server (8) containing a list of mailboxes to use for the remainder of the connection along with a shared secret allowing standard Phalanx communication to commence. Along with the response, the server will send packet fetch requests to the first several mailboxes to be used in preparation to receive further packets from the client.

The client uses the shared secret to determine the sequence of mailboxes the server expects the client to use and begins to send packets to these mailboxes. These data packets are paired with their corresponding requests and forwarded onto the server passing through the filtering ring by virtue of the holes opened by the requests. This constitutes the normal behavior of the Phalanx connection (9).

If at any point in time the server decides that the connection is no longer desirable or simply starts running low on resources, it can either decrease the rate at which

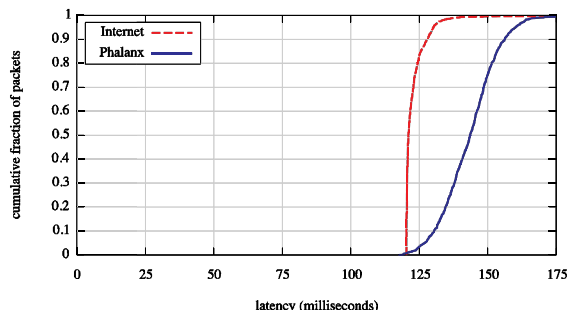


Figure 3: The cumulative distribution of round trip latencies over 1000 packets sent between Berlin `edi.tkn.tu-berlin.de` and North Carolina `planet02.csc.ncsu.edu` using both standard UDP (Internet) and Phalanx.

it requests new data packets or simply stops requesting packets altogether.

4 Evaluation

To evaluate Phalanx, we implemented a prototype server, client, filtering ring, and mailbox. Our prototype is approximately 1750 lines of C code between the four functions. We have not yet integrated our code with a scalable name system; several good candidates exist such as CoDoNs [28], DDNS [14], and Overlook [36], and integrating our system with one of these options is part of future work. We also augment the experimental results from our prototype with simulation results on a large-scale Internet topology.

4.1 Micro-benchmarks

To get a high-level idea of the performance of Phalanx in normal operating conditions with no ongoing attacks, we ran a series of experiments on PlanetLab. Each run sent packets between two randomly chosen PlanetLab nodes at different sites in late September 2007. (We eliminated PlanetLab sites where no node responded to a ping, and we avoided measuring during the time immediately preceding the conference submission deadline.) For each pair of nodes, we used iPlane to select an additional ten PlanetLab nodes, each at a physically different location, to serve as mailboxes. The nodes were selected as those that iPlane predicted would offer the lowest latency for a one hop path between the pair of nodes.

For each pair, we sent constant bitrate UDP traffic at approximately 25 kilobytes per second (25 packets per second) from the source to the destination first using Phalanx and then using plain UDP. In both cases we sent approximately 1000 packets per connection.

Triangle routing packets through mailboxes will usually incur some extra latency, particularly on PlanetLab where scheduling delays can dominate. However, iPlane's predictions can help counter this by selecting mailboxes which offer the least latency. Figure 3 shows a

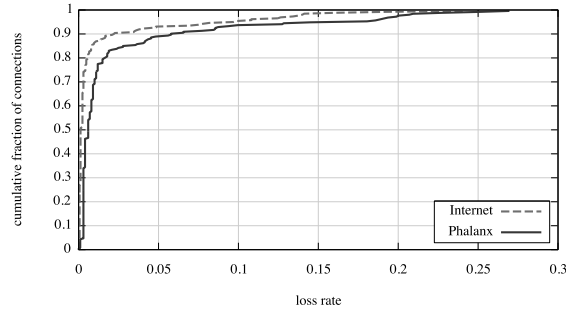


Figure 4: Cumulative distribution of loss rate, across all pairs of PlanetLab nodes, for standard UDP packets sent over the Internet and packets sent via Phalanx.

sample run for traffic between a node in Berlin and a node in North Carolina, measured for standard UDP and for Phalanx. As the figure shows, despite crossing the Atlantic, Phalanx has only a modest effect on per-packet latency. iPlane is able to find ten suitable mailboxes which do not significantly affect end to end latency. On average the latency goes up by a little less than 25 milliseconds or 20%.

A denser deployment of mailboxes, for instance using BitTorrent’s client network or Akamai’s worldwide network of servers, should improve these numbers significantly.

Because of its indirection architecture, Phalanx requires multiple packets to be successfully sent and received in order to successfully deliver what would be a single packet in the underlying Internet. As a consequence, we would expect that Phalanx’s packet loss rates would be worse than that of normal UDP.

Figure 4 illustrates this effect, showing the cumulative distribution of the measured loss rate across all pairs of PlanetLab nodes, for the experiment described above. While Phalanx does see more loss than the standard UDP, the effect is close to a constant factor increase in loss rate.

4.2 Attack Resilience

We next study Phalanx’s ability to counter attacks. For this, we repeat the previous experiment between the PlanetLab nodes in Berlin and North Carolina, but configured so that half of the mailboxes simulate an attack where they drop 75% of the arriving packets. This might occur if the mailboxes themselves were being flooded, or equivalently, if a portion of the filtering ring was being attacked. The sender and receiver respond to this loss by increasing the sending rate through the unaffected mailboxes to compensate; they could also expand the number of mailboxes in use, but this was not necessary to compensate for the simulated attack.

Figure 5 shows the result. After twelve seconds, the simulated attack begins. By increasing the sending rate, the receiver is able to maintain an average of 25 packets

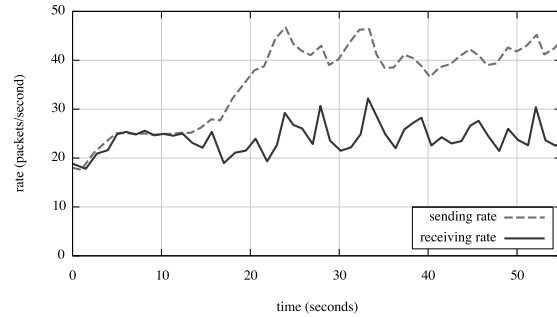


Figure 5: The rate at which packets are sent and received in the face of a modeled attack. Starting at 12 seconds, 5 of the 10 Phalanx mailboxes are “attacked”, cause 75% loss of arriving data packets. Noting the decrease in throughput, the receiver asks the sender to send faster and is able to maintain an average throughput of 25 packets per second.

per second despite half of the mailboxes dropping most of their packets.

4.3 Simulation

Evaluating systems like Phalanx at scale has always posed a problem because they are fundamentally intended to operate at scales well beyond what can be evaluated on a testbed. To address this issue, we built a simulator that captures the large-scale dynamics of Phalanx and allows us to simulate millions of hosts simultaneously.

The simulator uses a router-level topology gathered by having iPlane [22] probe a list of approximately 7200 known Akamai nodes from PlanetLab nodes. These Akamai nodes serve as stand-ins for appropriately located mailboxes. Each PlanetLab node serves as a stand-in for a server which is under attack.

We assume that attackers target the mailboxes, the server and the links near the server. Traffic is assumed to flow from clients to mailboxes unmolested. We assign link capacities by assuming mailbox access links are 10 Mbps, the server access link is 200 Mbps, and link capacity increases to the next category of {10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps, 40 Gbps} as the links move from the edge to the core.

We assign attackers with attack rates according to end-host upload capacity information gathered in our previous work [22, 18] and assume that good clients communicate at a fixed rate of 160 Kbps.

By using IP to AS mappings, we are able to simulate the behavior of the system under varying levels of deployment of the Phalanx filtering rings. Figure 6 shows the effect of increasing deployment of filtering rings for a server located at planetlab-01.kyushu.jgn2.jp. (The results are similar when we use other PlanetLab nodes as servers.) In this simulation, there are 100,000 attacking

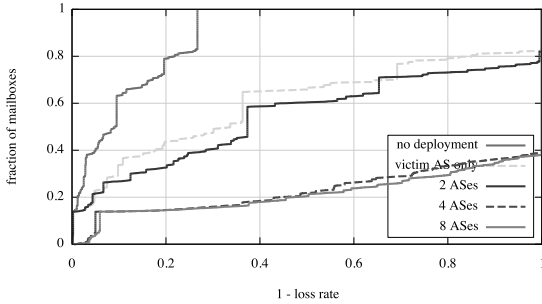


Figure 6: The cumulative fraction of mailboxes seeing at most a given fraction of goodput when communicating with the server.

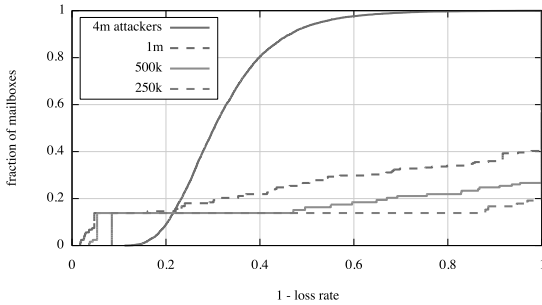


Figure 7: The cumulative fraction of mailboxes seeing at most a given loss rate for a varying number of attackers.

nodes and 1000 good clients all trying to reach the victim server. We simulate varying degrees of deployment by iteratively adding the largest adjacent AS to the current area of deployment.

As one might expect, even a little deployment helps quite a bit. Only deploying filters at the victim AS provides significant relief and allows some mailboxes to see lossless communication. Deploying in just 4 ASes (including the tier-1 AS NTT) results in the vast majority of mailboxes seeing lossless communication, effectively stopping the attack in its tracks if we assume that connections use any degree of redundancy to handle losses.

We next look at the scalability of Phalanx in handling attacks involving millions of bots. For this experiment we consider a somewhat stronger deployment: upgrading the mailboxes to 100 Mbps access links. Figure 7 examines the effect on mailbox loss rate as we increase the number of attackers. Most connections easily withstand the brunt of an attack involving one million nodes, and Phalanx still allows some (though severely degraded) communication through when facing 4 million nodes. In practice, Akamai advertises more than 15,000 nodes and many of them are connected via 1 Gbps links, which would easily give another order of magnitude in protection.

We are also able to quantitatively show the benefit of the multipathing approach in Phalanx over previous capability-based schemes like TVA [40]. Figure 8

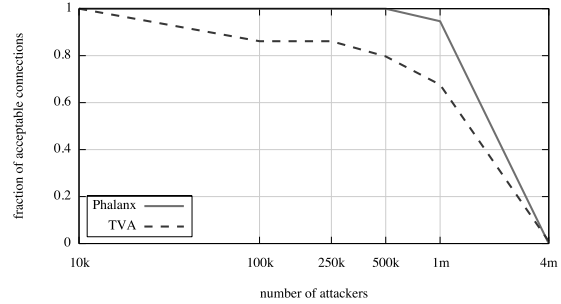


Figure 8: The fraction of connections which see less than 3% loss when sending 3 copies of each packet.

shows the fraction of connections which suffer acceptable loss when using a single-path system, like TVA, as compared with Phalanx. We use the same setup as the scalability experiment (mailboxes with 100 Mbps access links), with filtering rings being equivalent to TVA-enabled routers. Further we assume that all packets are sent three times to attempt to overcome loss. We define acceptable loss to be less than 3% after the redundancy.

The benefit of multipathing is immediately clear as Phalanx is able to provide almost all connections with acceptable loss rates even when defending against one million attackers. TVA on the other hand is forced to use a single path per connection meaning that some unfortunate connections will always have to endure high loss. Further, a calculating adversary can inflict the maximum possible damage with his available resources by targeting them on specific sets of links rather than being forced to spread his attack out.

5 Related Work

Our work on Phalanx leverages many elements from prior attempts to design a solution to mitigate DoS attacks. As we have argued, the primary differences come down to goals: ours is to design a system that could be easily deployed today, does not change the open model of Internet access, yet is powerful enough to deal with massive botnets.

5.1 Network Filtering

CenterTrack [35] and its later cousin dFence [24] propose new functionality for individual ASes to ensure that all traffic for a specific destination (e.g., one under attack) flows through certain filtering boxes. While effective for many attacks found in the wild today, this centralized approach is unlikely to scale to be able to handle the terabit attacks possible with multimillion-node botnets.

Pushback schemes [8, 9, 23] attempt to scale the power of network filtering by pushing filters to routers immediately downstream from where the traffic is entering the network. These filters are specially constructed to block attack traffic while allowing good traffic through. For example, in Active Internet Traffic Filtering [9], victims

alert their upstream routers of undesired flows. These routers temporarily block the flows while they in turn negotiate with routers further upstream to block the flows.

In the case of a multimillion-node botnet, where bots are found in almost every corner of the Internet, containing a botnet would require support for installing and managing filters at routers throughout the public Internet.

5.2 Capabilities

Capability schemes [7, 40, 10, 38, 39] have also received quite a lot of attention from the research community. Capability approaches drop (or relegate to lower priority) all traffic which does not carry a certificate proving that the packet was explicitly allowed by the recipient.

For example, in SIFF [39], capabilities are created during connection setup by each router on the path from source to destination. These routers stamp the initial connection request packet with time-limited cryptographic information. If the connection is desired by the destination, it returns the stamps to the sender, enabling it to prove to each router in the path that the connection is to be permitted. Because the capabilities are based on cryptographic hashes of flow identifiers, a secret key and local timestamps at routers, they cannot easily be forged or reused spatially or temporally.

The solutions for acquiring capabilities are noticeably less fleshed out and are usually some form of approximate fair queueing on the flow and/or path identifier. Furthermore, the capabilities are path-specific and any change to the route forces all flows using that path to reacquire their capabilities.

In Phalanx, we use a per-packet capability to identify which specific packets are to be allowed through the filtering ring at the request of a destination. Flows whose packets are not requested, do not have permission to send and are thus dropped. Additionally we make use of a more robust resource proof-based fair queueing scheme for the connection setup channel. Lastly, Phalanx uses loose source routing and rapidly refreshed capabilities to ensure that routing changes do not prevent the system from providing service even under attack.

5.3 Overlays

Overlay schemes [19, 5, 30, 33] leverage a set of trusted or semi-trusted nodes to act as proxies for end-hosts which may become the subject of attack. The common idea is to use more fully functional, and easier to deploy, machines to perform complex operations on packets, ones that might not be feasible at line rate inside of routers.

For example, one of the first overlay proposals for DoS protection was Secure Overlay Services (SOS) [19]. In SOS and later systems such as Mayday, only pre-authenticated users were allowed to route packets

through the overlay. Hence these systems could not be used for protecting general Internet traffic. A valid source could send packets to any node in the overlay by including an authentication token, and these packets were then routed to a specific output node (generalized later to be a set of output nodes), which then forwarded the packets to the destination. If the destination IP address and the identity of the overlay output node were both kept secret, an attack would be difficult to mount.

Phalanx builds on the basic approach of using an overlay network; in our view, overlays are the only way we will be able to deploy a solution to multimillion-node botnets in the foreseeable future. However, we generalize on the prior overlay work, so that in Phalanx: (i) any Internet host can send packets through the overlay without pre-authentication, (ii) any overlay node can send packets to any destination, and (iii) there is a simple, scalable, and efficient protocol for installing network filters to prevent unauthorized packets from reaching the destination.

5.4 Resource Proofs

A relatively new technique borrows resource proofs from techniques to deal with Sybil attacks [15]. Resource proofs allow service to be given in proportion to the resources available to a given user. This has the effect of preventing attackers from flooding and thus drowning-out well behaved users.

Speak-up [37] proposes using bandwidth for resource proofs by having legitimate hosts send more requests during times of attack. This results in per-bandwidth fair queue. OverDoSe [30] and Portcullis [26] both instead have hosts solve cryptographic puzzles to provide per-computation fair queueing.

In Phalanx, as in Portcullis, we use cryptographic puzzles to provide per-computation fair queueing because computation is a local-only, reasonably cheap resource especially in comparison to bandwidth.

5.5 Architectures

In addition, several proposals have suggested completely new architectures that would make denial of service attacks much harder to mount. For example, Off By Default [10] proposes that the network be modified to establish routes only where explicitly allowed by the recipient; in other words, only legitimate sources would be allowed to send packets to a specific destination. Establishing and tearing down routes is a fairly heavyweight operation, however, while Phalanx achieves the same goal on a per-packet granularity.

Phalanx is perhaps closest in spirit to the Internet Indirection Infrastructure (i3) proposal, in that every packet is sent through a mailbox abstraction [34]. Secure-i3 [4] extends the original i3 approach for routing via DHTs

to use an almost-unlimited address space to help prevent certain attacks. Like Phalanx, Secure-i3 makes use of a level of indirection in between sender and receiver as well as a large swarm computers acting as intermediaries. Unlike Phalanx requests, i3 triggers last for long periods of time, usually minutes, instead of being installed on a packet granularity. Further, i3 assumes that all packets are carried over i3, so that it cannot be attacked from below. Phalanx is designed to withstand attacks carried over the underlying Internet.

6 Conclusion

In this paper, we presented Phalanx, a system for addressing the emerging denial of service threat posed by multimillion-node botnets. Phalanx asks only for two primitives from the network. The first is a network of overlay nodes each implementing a simple, but carefully engineered, packet forwarding mechanism; this network must be as massive as the botnet that it is defending against. Second, we require a filtering ring at the border routers of the destination's upstream tier-1 ISP; this filtering ring is designed to be simple enough to operate at the very high data rates typical of tier-1 border routers. We have implemented an initial prototype of Phalanx on PlanetLab, and used it to demonstrate its performance. We have further demonstrated Phalanx's ability to scale to million node botnets through simulation.

7 Acknowledgments

We would like to thank Arun Venkataramani for a set of conversations which helped us realize the need for more scalable DDoS protection. We would also like to thank our shepherd, Sylvia Ratnasamy, as well as our anonymous reviewers for their help and valuable comments. Additionally, this work was supported by National Science Foundation grant #CNS-0430304.

References

- [1] Akamai technologies. <http://www.akamai.com/>.
- [2] Microsoft's unabated patch flow. <http://www.avertlabs.com/research/blog/index.php/category/security-bulletins/>, May 9 2007.
- [3] 'Surge' in hijacked PC networks. <http://news.bbc.co.uk/2/hi/technology/6465833.stm>, March 2007.
- [4] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Towards a more functional and secure network infrastructure. Technical report, UC Berkeley, 2003.
- [5] D. G. Andersen. Mayday: Distributed filtering for internet services. In *USITS*, 2003.
- [6] N. Anderson. Vint Cerf: one quarter of all computers part of a botnet. <http://arstechnica.com/news.ars/post/20070125-8707.html>, January 25 2007.
- [7] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. In *HotNets-II*, 2003.
- [8] K. Argyraki and D. Cheriton. Network capabilities: The good, the bad and the ugly. In *HotNets-IV*, 2005.
- [9] K. Argyraki and D. R. Cheriton. Real-time response to denial-of-service attacks. In *USENIX*, 2005.
- [10] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *HotNets-IV*, 2005.
- [11] D. J. Bernstein. SYN cookies. <http://cr.yp.to/syncookies.html>, 1996.
- [12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [13] M. Casado, P. Cao, A. Akella, and N. Provos. Flow-cookies: Using bandwidth amplification to defend against DDoS flooding attacks. In *IWQoS*, 2006.
- [14] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *IPTPS*, 2002.
- [15] J. R. Douceur. The sybil attack. In *IPTPS*, 2001.
- [16] P. Finn. Cyber assaults on Estonia typify a new battle tactic. <http://www.washingtonpost.com/wp-dyn/content/article/2007/05/18/AR2007051802122.html>, May 19 2007.
- [17] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *CCS*, 2007.
- [18] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Leveraging bittorrent for end host measurements. In *PAM*, 2007.
- [19] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *SIGCOMM*, 2002.
- [20] B. Krebs. Blue security kicked while it's down. http://blog.washingtonpost.com/securityfix/2006/05/blue_security_surrenders_but_s.html, May 2006.
- [21] A. Kuzmanovic and E. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants). In *SIGCOMM*, 2003.
- [22] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*, 2006.
- [23] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *CCR*, 2002.
- [24] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dFence: Transparent Network-based Denial of Service Mitigation. In *NSDI*, 2007.
- [25] A. McCue. Bookie reveals \$100,000 cost of denial-of-service extortion attacks. <http://software.silicon.com/security/0,39024655,39121278,00.htm>, June 11 2004.
- [26] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from Denial-of-Capability attacks. In *SIGCOMM*, 2007.
- [27] B. Raghavan and A. Snoeren. Decongestion Control. In *Hotnets-V*, 2005.
- [28] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM*, 2004.
- [29] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), Apr. 1992.
- [30] E. Shi, I. Stoica, D. Andersen, and A. Perrig. OverDoSe: A generic DDoS protection service using an overlay network. Technical report, Carnegie Mellon University, 2006.
- [31] E. Skoudis. Big honkin' botnet - 1.5 million! <http://isc.sans.org/diary.html?storyid=778>, October 2005.
- [32] S. Staniford, V. Paxson, and N. Weaver. How to Own the internet in your spare time. In *USENIX Security*, 2002.
- [33] A. Stavrou and A. D. Keromytis. Countering DoS attacks with stateless multipath overlays. In *CCS*, 2005.
- [34] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.
- [35] R. Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *USENIX Security*, 2000.
- [36] M. Theimer and M. B. Jones. Overlook: Scalable name service on an overlay network. In *ICDCS*, 2002.
- [37] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM*, 2006.
- [38] D. Wendlandt, D. G. Andersen, and A. Perrig. Bypassing network flooding attacks using fastpass. Technical report, Carnegie Mellon University.
- [39] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symposium on Security and Privacy*, 2004.
- [40] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, 2005.

Harnessing Exposed Terminals in Wireless Networks

Mythili Vutukuru, Kyle Jamieson, and Hari Balakrishnan
MIT Computer Science and Artificial Intelligence Laboratory
{mythili,jamieson,hari}@csail.mit.edu

Abstract

This paper presents the design, implementation, and experimental evaluation of CMAP (Conflict Maps), a system that increases the number of successful concurrent transmissions in a wireless network, achieving higher aggregate throughput compared to networks that use carrier sense multiple access (CSMA). CMAP correctly identifies and exploits exposed terminals in which two senders are within range of one another, but each intended receiver is far enough from the other sender that the two transmissions can succeed even if done concurrently. CMAP includes a reactive channel access scheme in which nodes transmit concurrently (even if there's the possibility of a collision), then observe the loss probability to determine whether they are better off transmitting concurrently or not. Experimental results from a 50-node 802.11a testbed show that CMAP improves throughput by $2\times$ over CSMA with exposed terminals, while converging to the performance of CSMA when the senders and receivers are all close to each other. CMAP also improves throughput by up to 47% over CSMA in realistic access point-based networks by exploiting concurrent transmission opportunities.

1 Introduction

It is well-known that maximizing the number of successful concurrent transmissions is a good way to maximize the aggregate throughput in a wireless network. Current contention-based channel access protocols generally attempt to minimize the number of packet collisions, allowing concurrent transmissions only when the nodes determine that they are unlikely to result in a collision. For example, in the popular *carrier sense multiple access* (CSMA) scheme, before transmitting, a sender listens to the channel and assesses whether a nearby node is transmitting. If no nearby node is transmitting, the sender transmits immediately. If a nearby node is transmitting, then the sender defers, waiting for some time after the end of the ongoing transmission. Then the sender repeats the same carrier sense-defer process.

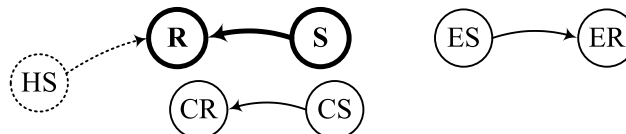


Figure 1. An example transmission from S to R with three abstract sender cases: an in-range but conflicting sender CS, an exposed sender ES, and a hidden sender HS.

Because a receiver's ability to decode a packet successfully depends on channel conditions near the *receiver* and not the sender, CSMA is at best a sender's crude guess about what the receiver perceives. This guess can be correct if the receiver and sender are close enough that they experience similar noise and interference conditions. However, it can also prevent a sender (e.g., ES in Figure 1) from transmitting a packet when its intended destination has a lower level of noise and interference—an *exposed terminal* situation. In addition, researchers have observed that receivers can often “capture” packets from a transmission even in the presence of interfering transmissions [18, 20], suggesting that simply extending the carrier sense mechanism to the receiver does not solve the problem. We argue that schemes like CSMA in which nodes use heuristics (such as “carrier is busy”) to perform channel access are too conservative in exploiting concurrency because they are “proactive”: nodes defer to ongoing transmissions without knowing whether in fact their transmission actually interferes with ongoing transmissions.

To improve throughput in a wireless network, we propose CMAP, a link-layer whose channel access scheme *reactively* and *empirically* learns of transmission conflicts in the network. Nodes optimistically assume that concurrent transmissions will succeed, and carry them out in parallel. Then, in response to observed packet loss, they discover which concurrent transmissions are likely to work, and which aren't (probabilistically), dynamically building up a distributed data structure containing a “map” of conflicting transmissions (e.g., S to R and CS to CR in Figure 1). In §3, we describe this novel *conflict map* data struc-

ture (hence the name CMAP), and show how nodes can maintain it in a distributed fashion by overhearing ongoing transmissions and exchanging lightweight information with their one-hop neighbors. By listening to ongoing transmissions on the shared medium to identify the current set of transmitters, and consulting the conflict map just before it intends to transmit, each node determines whether to transmit data immediately, or defer.

Of course, not all conflicting senders will be in range of each other to overhear and make the transmit-or-defer decision because of the well-known “hidden terminal” problem (**HS** in Figure 1). To prevent performance degradation in such cases, a CMAP sender implements a reactive *loss-based backoff mechanism* to reduce its packet transmission rate in response to receiver feedback about packet loss. Finally, note that any scheme that seeks to exploit the exposed terminal opportunity shown in Figure 1 must cope with link-layer ACKs from **R** to **S** being lost at **S** due to a collision with **ES**’s transmission. CMAP tolerates ACK losses with a *windowed ACK and retransmission protocol*.

We have implemented a CMAP prototype in software running on a 50-node testbed with commodity 802.11a wireless LAN hardware (§4). We present an evaluation of CMAP in §5 showing that CMAP leads to a $2\times$ improvement over CSMA with exposed terminals, while successfully avoiding interfering concurrent transmissions. In access point-based topologies with multiple concurrent transmissions, CMAP improves aggregate throughput by between 21% and 47%; the median per-source throughput is $1.8\times$ better than CSMA. CMAP also achieves a 52% improvement in aggregate throughput over CSMA in content dissemination mesh networks. These gains are mainly due to the non-interfering concurrent transmission opportunities that CMAP is able to exploit.

The contributions of this paper over existing proposals to solve the exposed terminal problem [1, 11, 16] are as follows. First, CMAP identifies all exposed terminal opportunities because it uses packet delivery probabilities, not heuristics that may (indirectly) influence packet delivery, to identify exposed terminals. Second, CMAP nodes gather the packet delivery probabilities in an online and distributed fashion, and do not require any offline measurements. Finally, CMAP demonstrates its gains in a live 802.11a testbed implementation.

2 Overview of CMAP

The CMAP design has three parts: a channel access (MAC) protocol, a windowed retransmission protocol, and a backoff mechanism that uses receiver feedback.

Channel access. The CMAP MAC uses a distributed data structure called the *conflict map*, which allows nodes to determine which pairs of transmissions are likely to obtain

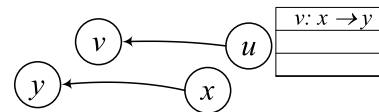


Figure 2. Example conflict map state shown for node u when it detects a transmission between x and y that conflicts with its transmission to v .

lower throughput if done concurrently than if done sequentially. The nodes in the network use *empirical* observations of packet losses to populate the conflict map, rather than assuming proactively (e.g., because the carrier is busy) that a given pair of transmissions shouldn’t be done concurrently because a collision might ensue.

Each node computes and stores a portion of the conflict map using feedback from its receivers about the fate of its transmissions. This conflict information at node u is a table with entries of the form $(v : x \rightarrow y)$. This entry, shown in Figure 2, means that if u were to send to v concurrently with a transmission from x to y , then the resulting throughput would be lower than if the two transmissions were done sequentially. We call such transmissions *conflicting*. If two transmissions conflict, it would be better for one of the senders (say, u) to *defer* its transmission while the other transmission is in progress. For this reason, we call this table the node’s *defer table*. The union of the defer tables of all the nodes in the network forms its conflict map.

Initially, the defer table at each node is empty, so nodes transmit without hesitation whenever they have data to send. Each receiver keeps track of the packet loss rates from its sender as well as what other concurrent transmissions were ongoing during the time it was receiving packets. If a receiver notices that the packet loss rate from its sender is high when another concurrent transmission is in progress, it infers that the concurrent transmissions conflict and propagates this information to the defer tables of the conflicting senders. §3.1 describes this process in detail.

Each node in the network continuously listens to the wireless channel to keep track of what other transmissions are currently in progress in its vicinity. When a node u is about to send a packet to node v , it consults its defer table to see if there are any entries of the form $v : x \rightarrow y$, such that there’s an ongoing transmission between x and y . If so, u defers its transmission until $x \rightarrow y$ completes and then re-attempts to transmit. Otherwise, it goes ahead and transmits. The transmission decision process is described in §3.2.

Windowed retransmission protocol. To increase the packet success rate observed by higher layers, receivers send link-layer ACKs for the received data packets; in response, the sender retransmits packets presumed lost. The CMAP retransmission protocol (§3.3) uses a window, un-

like current wireless LAN link layers that use a stop-and-wait retransmission protocol (i.e., a window size of 1). The ACKs sent by receivers are cumulative and contain a bitmap indicating which packets in the window have been received. The main benefit of the window mechanism is to avoid spurious retransmissions when only the ACK (and not the data packet) gets lost, thereby making the retransmission protocol resilient to ACK losses. This resilience is important for CMAP because although making transmission decisions using the defer table exploits exposed terminal opportunities, the ACKs have a high likelihood of being lost in collisions at the exposed senders. For example, in Figure 1, the ACK from receiver **R** to sender **S** can collide at **S** with a data transmission from **ES** to **ER**.

Backoff policy. As described thus far, for a sender to defer to an interfering transmission, the receiver must be able to identify the interferer and the sender must be able to overhear the interfering transmission. Therefore, CMAP may degrade performance when an interferer is out of hearing range of either the sender or the receiver; we show in §5.4 that the expected reduction in CMAP throughput due to such “hidden interferers” is around 10% of the link rate. To improve throughput in such cases, CMAP uses a loss rate-based backoff policy (§3.4). Because CMAP uses cumulative ACKs, senders in CMAP, unlike 802.11 senders, do not back off every time a transmission fails to elicit an ACK. Instead, receivers report the loss rate over a window of packets in every cumulative ACK, and senders back off when this loss rate exceeds a threshold.

2.1 Physical Layer Abstraction

CMAP encapsulates packets with a CMAP header and trailer before handing them over to the physical layer (PHY). We assume the following abstract model of the underlying PHY: it decodes and delivers the headers and trailers of received packets independent of the rest of the packet [5]. This PHY model has two important properties. First, it “streams” the header of an incoming packet to the CMAP layer before the rest of the packet reception completes. This property ensures that nodes learn of and defer to ongoing conflicting transmissions in a timely manner. Second, even if some bits in the packet’s payload are corrupted in a transmission, the PHY can salvage error-free headers and trailers and pass that information to CMAP. This ability to recover headers or trailers from a collision helps populate the conflict map (§3.1).

This abstract model of the physical layer can be realized in two ways. Our CMAP implementation (§4) uses a software “shim” that transmits separate small “header” and “trailer” packets with their own checksums (CRCs) before and after a data packet respectively; doing so provides the abstraction with the two properties mentioned above without modifying the current PHY implementations. An alter-

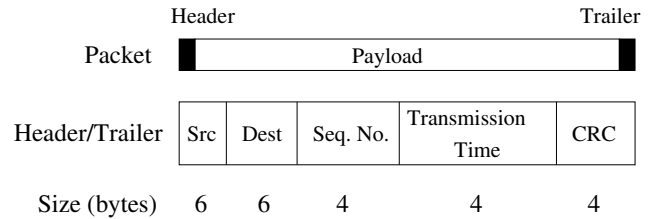


Figure 3. CMAP packet format.

nate approach, which requires hardware modification but has lower overhead, is to transmit the header and trailer as part of the packet and use recently-proposed partial packet recovery techniques [5] to decode headers and trailers independently. For ease of exposition and without loss of generality, however, we will describe the design of CMAP assuming the physical layer abstraction of the previous paragraph.

3 CMAP Design

This section describes CMAP’s design in detail. The CMAP packet format, header and trailer subfields and their suggested sizes are shown in Figure 3. In addition to the source and destination MAC addresses, the CMAP header and trailer contain the estimated transmission time of the packet, which lets deferring nodes decide how long they need to wait before attempting to send data. They also contain a link-layer sequence number and a separate CRC covering the entire header or trailer.

CMAP nodes are always in promiscuous mode, attempting to decode the headers and trailers of other concurrent transmissions that they can overhear.¹ For now, we assume that all packets are transmitted at a common bit-rate and power level. This assumption simplifies the discussion of the system; in §3.5, we discuss how CMAP must be modified to handle heterogeneous bit-rates and transmit power levels. We also assume that all transmissions are unicast; we discuss how to handle transmissions with more than one intended destination in §3.6.

3.1 The Conflict Map

We first describe how each node maintains its defer table to form the network’s conflict map. We use the notation $u \rightarrow *$ to denote a transmission from u to any other node (or to the broadcast address).

Each sender uses feedback obtained from receivers to populate its defer table. To provide this feedback, each receiver maintains an *interferer list* by observing the fate of packets reaching it, periodically broadcasting this list to all other nodes (senders) in its vicinity.

Constructing the interferer list. The interferer list at receiver node v , I_v , is a list of pairs (u, x) of sources u and in-

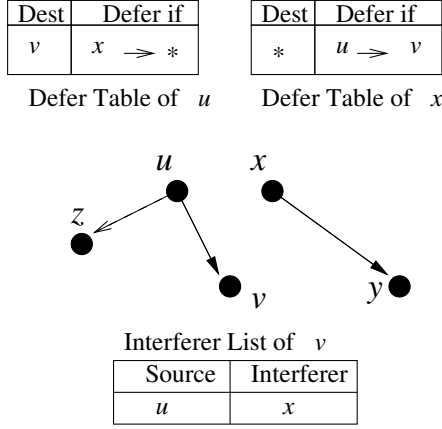


Figure 4. Example to illustrate CMAP's operation.

interferers x , such that $(u, x) \in I_v$ implies that the transmission $x \rightarrow *$ conflicts with the transmission $u \rightarrow v$ (see Figure 4 throughout this discussion). v adds this entry to the list after observing that transmissions from u to itself suffer a packet loss rate greater than a certain threshold l_{interf} whenever a transmission from x (to any other node) is concurrent; in such cases, it would make sense for u to defer its transmission to v when x was already transmitting data. Node v uses a threshold loss rate l_{interf} and not just a single packet loss to infer interference, because if x causes only mild interference to $u \rightarrow v$, then the overall throughput of $u \rightarrow v$ would be higher if the transmissions proceeded concurrently than if u waited for x to finish. In fact, one can see that as long as the loss rate observed at v is below 0.5, the throughput of $u \rightarrow v$ will be higher if u transmits concurrently with x than if u interleaves its transmissions with x 's transmissions. Therefore, l_{interf} must be 0.5.

To populate its interferer list, a receiver that experiences interference must know the identity of the interfering sender. The key insight here is that, in collisions of packets of comparable sizes, either the header or the trailer from each of the colliding senders can be salvaged with high probability (our physical layer delivers error-free headers and trailers). For example, we see in Figure 5 that when v 's reception of a packet from source u is corrupted by a collision due to an interfering transmission $x \rightarrow y$ that starts shortly afterward, v will be able to recover the header from u and the trailer from x .

When a receiver v detects a collision on a packet from u (by an unmatched header or trailer), it looks for headers or trailers from other sources received shortly before and after the collision. The receiver can verify that the transmissions corresponding to such headers or trailers actually overlapped in time with its reception using the transmission time information in the headers and trailers. When v identifies an interfering source x in this manner, it adds the pair (u, x) to I_v if the packet loss rate from sender u is above the

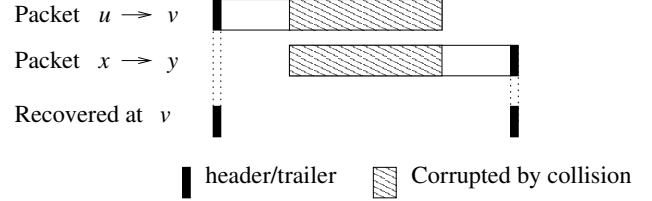


Figure 5. One of header or trailer of a packet usually survives in a packet collision.

threshold l_{interf} . Entries in the interferer list are timed out periodically to accommodate changing channel conditions and interference patterns.

Nodes periodically broadcast their interferer lists to all neighbors, either as standalone packets or by piggy-backing the interferer lists with routing beacons or other control messages. In general, it suffices to broadcast the interferer list to just the one-hop neighbors, because a receiver does not hear headers from interferers that are more than a hop away and hence does not know about them. However, in networks with asymmetric links (e.g., the receiver can hear the interferer's headers but the interferer cannot hear the receiver's interferer list updates), it may help to propagate the interferer list over two hops.

Populating the defer tables. When a node \mathcal{P} receives an interferer list I_r from node r , it updates its defer table using the following two *local* rules at \mathcal{P} :

Update rule 1: $\forall q : (\mathcal{P}, q) \in I_r$ add $(r : q \rightarrow *)$ to the defer table.

Update rule 2: $\forall q : (q, \mathcal{P}) \in I_r$ add $(* : q \rightarrow r)$ to the defer table.

To understand the above rules, consider again the example shown in Figure 4. When u receives v 's interferer list, it adds the entry $(v : x \rightarrow *)$ to its defer table by Rule 1, because u now knows that its transmitting packets to v while x is transmitting to *any* node is likely to cause a high packet loss at u .² Note that u need not defer while transmitting to all destinations, e.g., it may be able to transmit successfully to some other node z while $x \rightarrow *$ is in progress. Accordingly, Rule 2 does not apply at u .

On the other hand, when x receives v 's interferer list, it adds an entry $(* : u \rightarrow v)$ to its defer table by Rule 2. Note that x cannot transmit to *any* destination (not just y) while $u \rightarrow v$ is in progress, because its transmission to any node will cause interference at v . On the other hand, x can transmit freely when u is transmitting to a node other than v (say, z) as long as it knows it doesn't cause interference at that node. Accordingly, Rule 1 does not apply at x .

3.2 Transmission Decision Process

Each node keeps track of all ongoing transmissions it has heard about in the *ongoing list*, using the source, destination, and transmission time fields of the packet header to add and expire entries from this list. Suppose node u wants to send a packet to destination v . First, u checks that v is neither sending nor receiving packets at that moment. Next, for each communicating pair $p \rightarrow q$ in the ongoing list, u checks its defer table to see whether one of its entries matches the following patterns that indicate a conflict between the two transmissions:

Defer pattern 1: $(* : p \rightarrow q)$

Defer pattern 2: $(v : p \rightarrow *)$

If no match exists, then u immediately sends the packet to v . Otherwise, it defers its transmission until the matching transmission ends, waits a small amount of time ($t_{deferwait}$) to check if any other transmission begins, then conducts the same check again (it can't simply send because some other entry could now match).

One further optimization to this process is to send a non-conflicting packet to *another* destination, if the current packet at the head of the queue must be deferred. This optimization requires per-destination queues, but is not hard to implement, though some care must be taken to ensure that no queue is starved. We believe that this scheme will further improve throughput, and plan to evaluate it in the future.

3.3 Windowed ACK and Retransmission Protocol

After transmitting a packet, a sender waits for an ACK from the receiver for up to a duration $t_{ackwait}$. If the ACK does not arrive in this duration, the sender does not immediately retransmit the packet. It instead transmits a *send window* of up to N_{window} unacknowledged packets. This window mechanism helps avoid spurious retransmissions in the case when the packet was correctly received at the destination but the ACK gets lost due to interference at the sender. We use $N_{window} = 8$ packets to tolerate a significantly higher loss rate on ACKs than on the data packets.

The ACKs sent by receivers upon every successful packet reception are cumulative and include a bitmap indicating which packets in the send window were received correctly. The ACKs also include the packet loss rate perceived by the receiver over the previous window of packets, computed using the link-layer sequence numbers in packet headers and trailers.

When the number of unacknowledged packets $N_{outstanding}$ at a sender reaches the maximum N_{window} , the sender times out for a random time chosen between the minimum timeout τ_{min} and maximum timeout τ_{max} before retransmitting each unacknowledged packet in its window in sequence. Because the absence of an ACK for the entire

window may indicate extreme interference at either the sender or the receiver, τ_{max} should be at least as long as the time taken to transmit an entire window's worth of packets in order to allow the interfering transmission at the destination to complete. We pick $\tau_{max} = \frac{N_{window} \text{ (bits)}}{\text{link speed (bits/s)}}$ and $\tau_{min} = \frac{\tau_{max}}{2}$.

3.4 Backoff Policy

CMAF's ability to prevent conflicting transmissions depends on the sender and receiver being able to overhear the headers and trailers from an interfering transmission. For example, the conflict map algorithm may not work when the conflicting senders cannot hear each other (the hidden terminal problem) or when the receiver is not in the hearing range but experiences interference from a far-away interferer (see §5.4 for an evaluation of CMAF in such cases). Nodes also experience transient losses before the feedback from receivers propagates to the defer tables of the senders. In order to slow down the senders and limit losses in such cases, CMAF implements the following backoff mechanism between consecutive packet transmissions at senders. Nodes maintain a contention backoff window CW like 802.11. After transmitting a packet and waiting for an ACK for up to a duration of $t_{ackwait}$, nodes also wait for a random backoff duration between 0 and CW before attempting to transmit the next packet.

Because CMAF uses windowed transmissions unlike the 802.11 MAC, the sender updates CW in response to the packet loss rates reported in ACK packets and not in the absence of ACKs after a packet. CMAF senders update their contention backoff window CW upon receiving an ACK as follows. If the loss rate reported in the ACK is below a threshold $l_{backoff}$ (0.5 in our implementation; we found in our evaluation that CMAF's performance was not sensitive to the choice of the threshold), the sender resets its CW to zero. If the loss rate is above the threshold, the sender increments CW to a nonzero value, starting with the minimum CW_{start} and doubling it on every consecutive increment, up to a maximum CW_{max} . CW_{start} and CW_{max} are chosen to roughly mirror the corresponding 802.11 values. Note that the sender does not update CW when an ACK does not arrive between packets of a send window to avoid unnecessary backoffs in response to just ACK losses. Thus the backoff in CMAF is more resilient to ACK loss than the backoff in 802.11. Figures 6 and 7 summarize the pseudocode for transmitting a packet and processing ACKs in CMAF.

3.5 Handling Multiple Bit-rates

So far in our discussion, we have assumed that all transmissions are performed at a common bit-rate and power level. To handle heterogeneous bit-rates, nodes annotate the interferer lists and defer tables with the bit-rates of the sources when the interference occurred. The decision to

```

while data to send and  $N_{outstanding} < N_{window}$  do {
  while defer table does not permit do {
    wait until end of current transmission +  $t_{deferwait}$ 
  }
  transmit and add packet to sent queue
  wait up to  $t_{ackwait}$  for an ACK
  wait for a backoff duration  $\in [0, CW]$ 
}

```

Figure 6. Pseudocode for sending a packet.

```

clear packets covered by ACK from sent queue
 $N_{outstanding} \leftarrow N_{out}$  - number acked packets
update loss rate estimate  $l$  from ACK
if ( $l > l_{backoff}$ ) then {
  if  $CW = 0$  then
     $CW \leftarrow CW_{start}$ 
  elseif  $CW < CW_{max}$  then
     $CW \leftarrow 2 \cdot CW$ 
}
else
   $CW \leftarrow 0$ 

```

Figure 7. Pseudocode for processing ACKs.

transmit at a node will then depend on the defer table entry corresponding to the bit-rate of its intended transmission and of the ongoing transmission. The extensions to handle multiple power levels are similar. We have not yet incorporated these bit-rate extensions in our implementation.

In fact, online bit-rate adaptation algorithms can benefit from using the information in the conflict map in choosing the best rate at which to transmit. For example, a node may choose to transmit at a lower rate that can tolerate interference from an ongoing transmission or defer to the ongoing transmission and transmit at a higher rate later on, picking the choice that yields a higher throughput. A preliminary evaluation of CMAP at higher bit-rates (§5.8) indicates that such an algorithm would amplify CMAP’s gains.

3.6 Beyond Unicast Transmissions

Thus far, we have assumed that all transmissions are unicast. However, senders may also transmit broadcast packets that are intended to reach some or all of the node’s one-hop neighbors. To make channel access decisions, such broadcast transmissions could simply be treated as a collection of unicast transmissions. For example, suppose a node u wishes to make a broadcast to a set of nodes \mathcal{V} . To make a decision on whether to transmit or not, u uses the transmission decision process in §3.2 to check that, $\forall v \in \mathcal{V}$ and for every ongoing transmission $p \rightarrow q$, $u \rightarrow v$ does not conflict with $p \rightarrow q$. In opportunistic routing [2] however, senders leverage broadcast transmissions in a slightly dif-

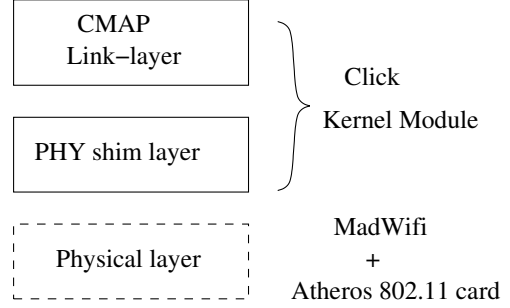


Figure 8. Architecture of the CMAP prototype; the components shown in solid lines were implemented by us.

ferent way—the sender broadcasts a batch of packets to a set of possible “forwarders” that opportunistically receive and forward some fraction of the packets each to the destination. In such broadcasts, it is sufficient to ensure that a packet is correctly received at one of the forwarders, not all. To handle such broadcasts, the conflict map data structure described in this section must be augmented with packet reception rates at receivers in the presence of interference. The sender’s decision on whether to transmit or not will then be based on the probability that at least one forwarder receives the packet, given the ongoing transmissions. Adapting CMAP to handle opportunistic routing is future work.

4 Implementation

In this section, we describe our implementation of CMAP (summarized in Figure 8) and quantify its overheads. We have implemented CMAP using the Click [9] router kernel module on Linux, and commodity 802.11 hardware driven by MadWifi [10]. To control channel access and retransmissions from the CMAP kernel module, we disabled carrier sense, link-layer ACKs, retransmissions and 802.11 backoff in the wireless card. All the nodes run in the promiscuous (“monitor”) mode of the MadWifi driver.

4.1 Adaptations For a Software MAC

We now describe the challenges in deploying and evaluating CMAP on commodity wireless hardware, and the adaptations in our implementation to overcome these challenges. First, the current 802.11 physical layer delivers headers of a packet along with the complete packet only after packet reception has completed, and headers and trailers from a corrupted packet cannot be recovered. In order to provide a streaming physical layer abstraction (§2.1) to the link-layer, our implementation uses a “shim” layer that transmits separate “header” and “trailer” packets immediately before and after a data transmission respectively. We

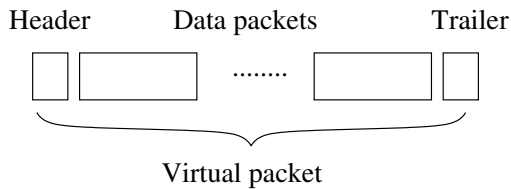


Figure 9. Virtual packet format in the CMAP prototype.

will refer to the header, data, and trailer packets together as a “virtual packet”. The shim is implemented in Click and is located between the link and physical layers.

Second, the gap between the end of a data transmission and arrival of the corresponding ACK is high in our implementation due to the communication latency between the software MAC and the hardware physical layers at both the sender and receiver. For example, in a typical experiment, this gap was observed to be between 0.5 and 2 milliseconds for about 90% of the data packets, and between 2 and 5 milliseconds for the rest. This latency is excessive because it takes only about 2 ms to transmit a 1400-byte packet at the lowest data rate of 6 Mbits/s in 802.11a. This latency also applies to received headers and trailers, and may prevent senders from processing the received headers of conflicting transmissions before transmitting data.

To overcome these limitations, our implementation sends N_{pkt} data packets destined to the same node between a header and trailer in one virtual packet, as shown in Figure 9. This approach effectively amortizes the cost of waiting for an ACK over multiple data packets, and allows senders to react in a timely manner to concurrent transmissions. In this implementation, defer and backoff decisions are made once per virtual packet; once a decision to transmit a virtual packet is made, the header, trailer and all the data packets are sent without any gap in between. The receiver sends an ACK after receiving the trailer of a virtual packet; the bitmap contained in the ACK acknowledges the receipt of individual data packets within a virtual packet.

4.2 Choosing Values For Design Parameters

We now discuss the implementation choices for the various design parameters of CMAP. Our implementation uses $t_{deferwait} = t_{ackwait} = 5$ ms to accommodate the propagation delay between the link and physical layers, as measured in §4.1. We use $N_{pkt} = 32$ in our implementation because such a CMAP implementation has comparable performance to the commodity 802.11 protocol—the single link throughput of CMAP at 6 Mbits/s is 5.04 Mbits/s while that of 802.11 with link-layer ACKs is 5.07 Mbits/s—enabling a fair comparison of CMAP and 802.11. The send window of unacknowledged packets is set to $N_{window} = 8$ virtual packets, or 256 data packets. The contention window parameters

CW_{start} and CW_{max} are set to the corresponding 802.11 values scaled by N_{pkt} —5 ms and 320 ms respectively.

5 Evaluation

The goal of this section is to measure CMAP’s ability to improve wireless network throughput by increasing the amount of spatial reuse. To this end, we compare the performance of our CMAP software prototype (described in §4) to that of the 802.11 MAC with carrier sense enabled (the “status quo”), and 802.11 with carrier sense disabled. We summarize our results below.

- In experiments with two pairs of senders and receivers, CMAP successfully exploits concurrent transmission opportunities to achieve up to $2\times$ improvement over 802.11 with carrier sense when the nodes are in an exposed terminal situation (§5.2), while effectively avoiding interfering concurrent transmissions using the conflict map data structure (§5.3). CMAP’s windowed retransmission protocol is central to realizing the full throughput gain in exposed terminal cases.
- The impact of “hidden interferers” that are out of communication range of either the sender or receiver on CMAP throughput is small (§5.4), and CMAP’s back-off scheme ensures no performance degradation compared to the status quo in such cases (§5.5).
- In realistic access point-based topologies with multiple concurrent senders, CMAP improves aggregate throughput by between 21% and 47% and median per-source throughput by $1.8\times$ compared to the status quo by exploiting exposed terminal opportunities (§5.6).
- CMAP improves throughput by 52% compared to the status quo in two-hop content dissemination mesh topologies (§5.7).
- CMAP’s performance benefits apply across multiple bit-rates (§5.8).

5.1 Experimental Testbed and Method

Our testbed consists of Soekris net4526 computers with a 133 MHz 486 processor running the 2.4.26 Linux kernel. The nodes are equipped with an 802.11 a/b/g mini-PCI card based on the Atheros AR5212 chipset.

The testbed nodes are located in one large floor of an office building as shown in Figure 10. Transmitting in isolation, of the 2162 node pairs that have any connectivity whatsoever, approximately 68% links have a packet reception rate (PRR) less than 0.1, 12% have a PRR greater than 0.1 and less than 1, and 20% have a PRR of 1. Considering just the latter two types of links, the nodes in our testbed have a mean degree of 15.2 and a median degree of 17.

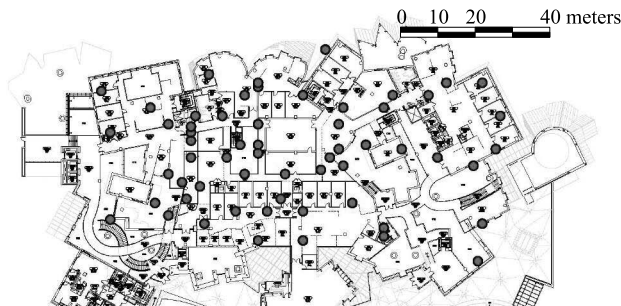


Figure 10. A map of our 50-node indoor wireless testbed.

We perform all our experiments in the 5 GHz 802.11a band which had negligible background traffic; we leave an evaluation of CMAP in other frequency bands with different propagation characteristics and potentially more non-CMAP background traffic to future work. Unless mentioned otherwise, all senders transmit 1400-byte data packets at the 6 Mbits/s bit-rate of 802.11a as fast as they can. Each run of an experiment lasts for 100 seconds and the aggregate throughput achieved is measured as the total number of non-duplicate data packets received per second at the designated receivers, computed over the last 60 seconds of the experiment.

We pick sender-receiver pairs in each experiment based on measurements of PRR and average signal strength between pairs of nodes at 6 Mbits/s and in the absence of any other concurrent transmission, obtained shortly before running the corresponding experiment. In all experiments below, we define two nodes a and b to be “in-range” of each other if both the links $a \rightarrow b$ and $b \rightarrow a$ have a PRR above 0.2 and signal strength above the 10th percentile of the signal strength of all links network-wide. We call a link $a \rightarrow b$ a “potential transmission link” if both the links $a \rightarrow b$ and $b \rightarrow a$ have a PRR above 0.9 and signal strength above the 10th percentile of the signal strength of all links network-wide; we pick only such links to send data in our experiments because any routing protocol running on this testbed typically selects such links. Note that while the PRR of a link alone could serve as a good metric to decide whether the link is a potential transmission link or not, we also use a low signal strength threshold to eliminate very weak links whose performance would degrade precipitously in the presence of other transmissions in the network.

5.2 CMAP Exploits Exposed Terminals

In this experiment we seek to quantify the maximum throughput gain two simultaneous wireless data streams can achieve in an exposed terminal configuration. Since this situation occurs frequently in busy access point-based wire-

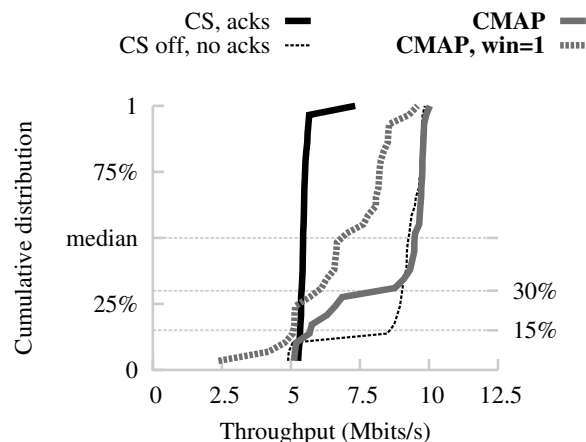


Figure 12. Experiment with exposed terminals. CMAP achieves a $2\times$ gain over 802.11 with carrier sense.

less networks [6], we expect the gains we find in the results to be applicable to such networks.

We pick pairs of links from our 50-node testbed, as shown in Figure 11(a), such that: (i) the two senders are in range of each other, (ii) each sender-receiver pair is a potential transmission link (as per the definitions in §5.1), (iii) the signal strength from a sender to its corresponding receiver is strong (in the 90th percentile of signal strength of all links network-wide), and (iv) the signal strength between all other pairs of nodes is somewhat weak (below the 90th percentile threshold). The last two constraints ensure that the two transmissions do not interfere very much, most likely resulting in an exposed terminal situation.

Figure 12 presents the distribution of throughput across 50 exposed terminal configurations chosen at random from all possible configurations. With carrier sense enabled, we see that most link pairs achieve little more than the single-link rate of 5 Mbits/s, since most of the time, each sender defers its transmission to the other.

With carrier sense and link-layer ACKs disabled (thin dashed curve), we see that 15% of the pairs are not in fact exposed terminals in the sense that the two transmissions do not simultaneously achieve their maximum throughput. Of the remaining 85% of pairs in this experiment, CMAP permits the two transmissions to proceed concurrently $\frac{70\%}{85\%} = 82\%$ of the time, resulting in a throughput improvement of approximately $2\times$ over 802.11 with carrier sense enabled. By carefully scrutinizing the experiment logs, we verified that the remaining 18% of pairs experienced many spurious retransmissions due to very high ACK loss rates that our windowed ACK scheme could not completely eliminate.

To quantify the benefits of our windowed retransmission protocol, we repeated the CMAP experiments with a window size of one virtual packet. We found that the median throughput improvement in this case was just $1.5\times$, signif-

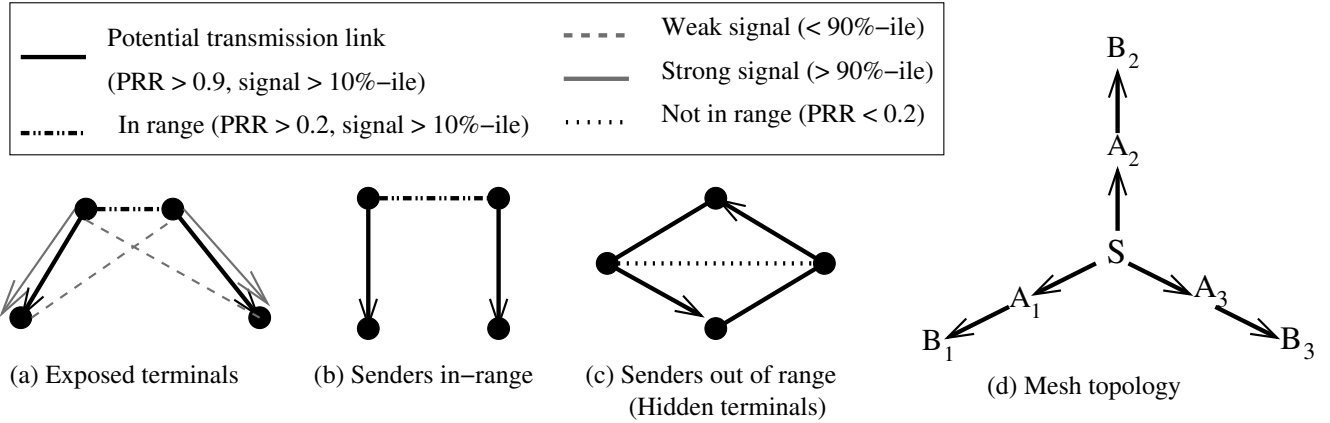


Figure 11. Constraints on topologies in experiments with exposed terminals (§5.2), two senders in-range (§5.3) and out of range (§5.5), and mesh topologies (§5.7). All links that are not annotated in the figure are unconstrained.

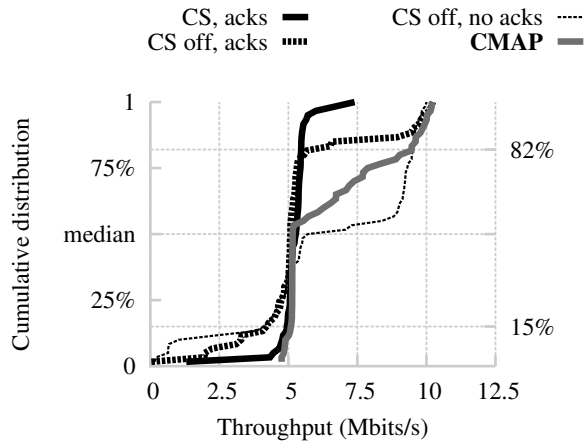


Figure 13. Experiment with two senders in range of each other. CMAP correctly identifies interfering concurrent transmissions.

icantly lower than CMAP with a window of eight virtual packets, because ACKs collided frequently at the senders and caused the senders to timeout and perform spurious re-transmissions.

5.3 CMAP Avoids Interfering Transmissions

We now evaluate CMAP on more general two-sender topologies. In this experiment, we evaluate the performance of CMAP on a topology with two sender-receiver pairs such that the two senders are in hearing range of each other. This experiment tests whether the defer scheme correctly discriminates between interfering and non-interfering concurrent transmissions.

We choose two sender-receiver pairs for this experiment as shown in Figure 11(b): the two senders are in range of

each other, and each sender-receiver pair is a potential transmission link; unlike in the experiments with exposed terminals (§5.2), we impose no additional constraints on the signal strengths of the links. Note that some pairs of links could well be exposed terminals.

Figure 13 presents the distribution of throughput across 50 link pairs chosen at random. On 15% of the link pairs, simultaneous transfers were deleterious, evidenced by the worse performance of 802.11 with carrier sense disabled compared to 802.11 with carrier sense enabled. For these link pairs, CMAP correctly directs that each transmission defer to the other and tracks the performance of 802.11 with carrier sense on (5 Mbits/s). However, there are link pairs in this experiment that are better off transmitting concurrently (18% of them on the right-hand side of the CDF), for which 802.11 with carrier sense and link-layer ACKs disabled offers a throughput improvement up to 2×. For these link pairs, CMAP correctly directs transmissions to occur simultaneously, and thus achieves roughly the same throughput improvements as 802.11 with carrier sense disabled. Also note that, over link pairs that are on the right side of the CDF, 802.11 with carrier sense disabled performs worse than CMAP when link-layer ACKs were enabled because 802.11 uses a stop-and-wait transmission method (unlike CMAP’s windowed retransmission scheme) and hence is more vulnerable to the ACK loss problem.

5.4 How Bad Are Hidden Interferers?

CMAP’s conflict map and defer mechanisms can fail to detect conflicting transmissions when either (a) the receiver is unable to receive at least *some* headers and trailers from an interferer in order to populate its interferer list, and thereby the conflict map, or (b) a potential sender cannot hear the interferer’s transmission headers in order to defer to it (the hidden terminals problem). As a result, an interferer

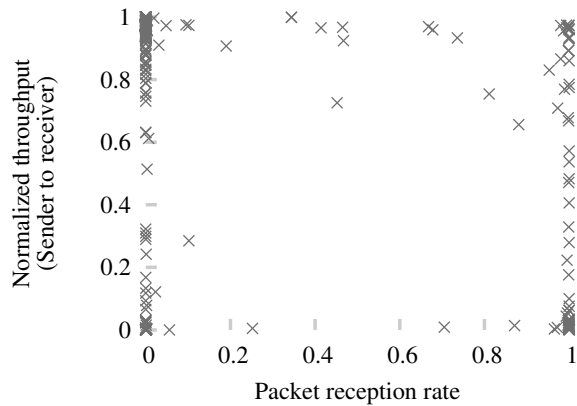


Figure 14. Scatter plot of the normalized throughput of a pair $S \rightarrow R$ in the presence of an interferer I vs. the minimum of the packet reception rates of the links $I \rightarrow R$ and $I \rightarrow S$.

that is out of communication range of either the receiver or the sender of a transmission (a “hidden interferer”) can degrade the throughput of that transmission. In this section, we evaluate the frequency of hidden interferers and their impact on CMAP throughput.

To identify the frequency of hidden interferers, we quantify the relationship between an interferer I ’s effect on the throughput of a transmission $S \rightarrow R$ and the probability that S and R can hear I . In this experiment, we randomly choose 500 pairs of sender-receiver pairs (S, R) with a potential transmission link between them, and for each pair, we pick an interferer I at random from all the nodes in the testbed. We first measure the throughput of the link $S \rightarrow R$ and the PRRs of the links $I \rightarrow R$ and $I \rightarrow S$ in the absence of any other interference. S and I then transmit 802.11 packets continuously and simultaneously with carrier sense and link-layer ACKs disabled³, and we measure the throughput of $S \rightarrow R$ in the presence of I ’s transmissions.

Figure 14 shows a scatter plot of the normalized throughput of $S \rightarrow R$ with interference (the ratio of the throughput of $S \rightarrow R$ in the presence of I ’s interference to that in the absence of interference) on the Y-axis, and the minimum of the packet reception rates of $I \rightarrow R$ and $I \rightarrow S$ on the X-axis. Note that the data points that lie near the left bottom of this graph correspond to hidden interferers, because for such points I reduces the throughput of $S \rightarrow R$ and yet is out of communication range of either S or R . From the data, we find that the fraction of points that lie in the left bottom quadrant of the plot (i.e., the fraction of cases where I reduces the throughput of $S \rightarrow R$ by more than 50%, but has a link with PRR less than 0.5 to either S or R) is only 8%. This observation convinces us that hidden interferers do not occur very frequently in our experiments.

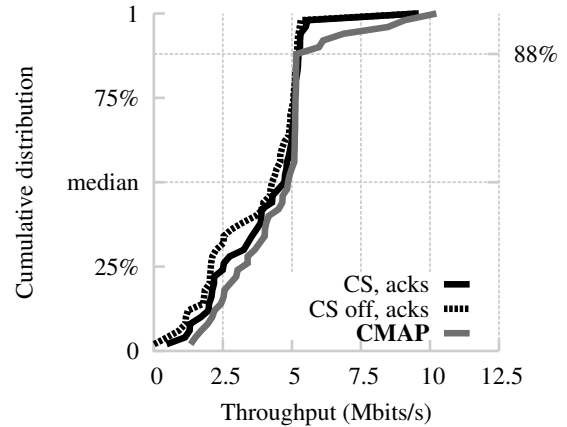


Figure 15. Experiment with two senders out of each other’s transmission range. CMAP’s backoff strategy avoids performance degradation compared to status quo.

We now estimate the magnitude of impact a hidden interferer has on CMAP throughput as follows. Let p_r and p_s denote the packet reception rates of the links $I \rightarrow R$ and $I \rightarrow S$ respectively. Then the probability that both S and R hear I is at least $p = \max(p_r + p_s - 1, 0)$. Let T denote the normalized throughput of $S \rightarrow R$ in the presence of I . Now, had the experiment been run with CMAP, the conflict detection mechanism would have avoided a throughput degradation (i.e., resulted in a normalized throughput of 1) with a probability p , and resulted in a lower throughput of T with a probability $1 - p$. Hence, the expected throughput of $S \rightarrow R$ with CMAP can be computed as $p \cdot 1 + (1 - p) \cdot T$. A sum of the above expression over all data points in Figure 14 works out to be 0.896. Thus, the expected damage to a CMAP pair’s throughput due to a hidden interferer is only around 10%. In practice, however, the degradation will be even smaller (as we will see next) because CMAP senders back off in response to losses, unlike the senders in the above experiment which were made to transmit continuously.

5.5 CMAP Handles Hidden Terminals Well

We now evaluate how well CMAP’s backoff protocol prevents performance degradation when the defer mechanism fails in an experiment with pairs of hidden terminals. We choose pairs of links for this experiment as shown in Figure 11(c): each receiver has a potential transmission link to both senders (as per the definitions in §5.1), ensuring that the two transmissions will almost always interfere with each other at the receivers. The senders are not in range of each other with the result that they cannot defer to each other’s transmissions.

Figure 15 presents the distribution of throughput across 50 randomly chosen link pairs. We see that CMAP and

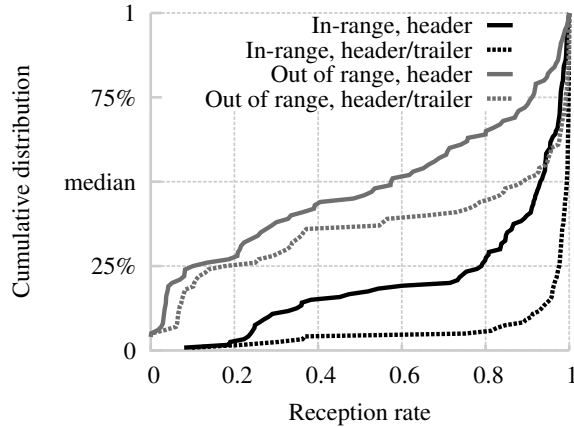


Figure 16. Probabilities of reception of either header or trailer and header alone for each transmitted virtual packet, computed from the experiments with two pairs of senders in-range (§5.3) and out of range (§5.5).

802.11 (with both carrier sense enabled and disabled) perform comparably. Also note that there is very little weight on the right-hand side of the CDF that represents throughputs greater than the single pair throughput. This is because the best we can hope for in such topologies, with current 802.11 hardware, is transmissions interleaved with each other to achieve the throughput of a single sender-receiver pair.

We also use the above experiment to validate our design decision of transmitting both headers and trailers (as opposed to only headers) on packets. For each experiment with two senders in §5.3 and §5.5, we compute the fraction of virtual packets transmitted by a sender for which either the header or the trailer was successfully received by the receiver. We compare this fraction against the fraction of virtual packets for which the header was received. We plot a CDF of these fractions across all sender-receiver pairs in each experiment in Figure 16. We see that the probability of reception of a header or trailer is higher than the probability of reception of a header alone in both the experiments; the benefit of using trailers is more pronounced when the senders were out of each other’s range and persistently collided at the receivers. We also observe that the probability that either a header or trailer is received is almost 1 in the experiment where the senders are in range of each other and transmit equal-sized packets.

5.6 Access Point Topology

In this section, we evaluate CMAP on larger topologies with multiple concurrent senders. We pick topologies that resemble wireless local area networks (WLANs) with multiple access points (APs) and clients that span a geographical area several radio-ranges in diameter.

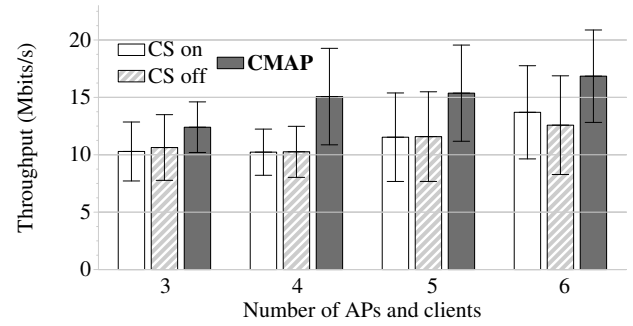


Figure 17. Mean throughput in the experiment with N APs and N clients; error bars show standard deviation. CMAP achieves between 21% and 47% more throughput than the status quo.

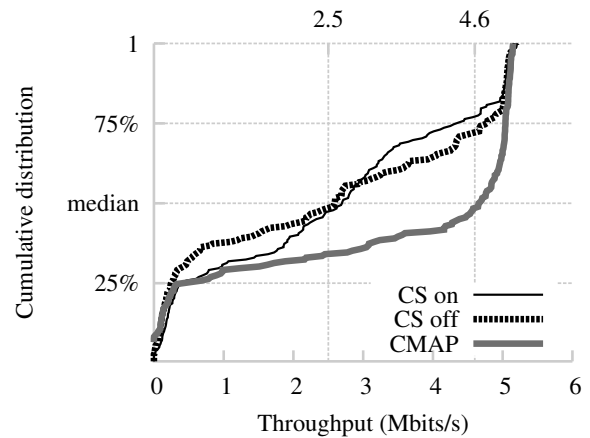


Figure 18. CDF of the per-sender throughput in experiments with N APs and N clients. CMAP increases the median by a factor of 1.8.

We divide the testbed (see Figure 10) into six “regions” and designate one node in each region as an AP, such that each AP is out of the communication range of every other AP. We choose clients of an AP from the set of nodes in that region that have a potential transmission link to that AP, and randomly designate one of the client or AP as the sender of packets. We then perform experiments by varying the number of APs (N) from three through six, always choosing APs from adjacent regions when there are fewer than six APs in an experiment. For each value of N , we perform 10 experiments with different clients for APs each time.

Figure 17 shows the average aggregate throughput of CMAP and 802.11 (with both carrier sense enabled and disabled) as a function of the number of concurrent senders N in the experiment. We find that CMAP improves aggregate throughput by between 21% (when $N = 3$) and 47% (when

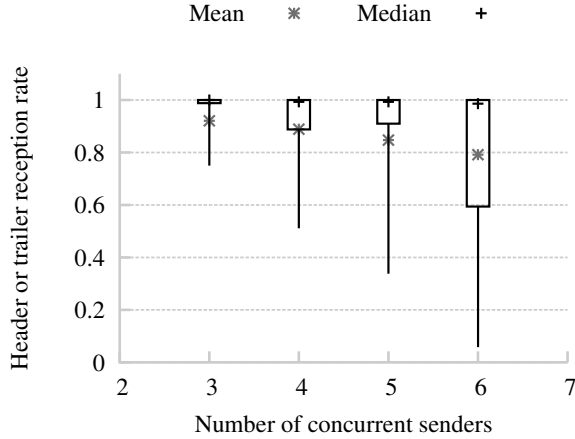


Figure 19. Mean and median probabilities of reception of either header or trailer of a virtual packet at a receiver, as a function of the number of concurrent senders in the experiment. The thick error bars represent the 25th and 75th percentiles, and the thin error bars show the 10th and 90th percentile values.

$N = 4$). CMAP sees this improvement because pairs of senders in adjacent cells were often exposed terminals. Figure 18 shows a CDF of the per-sender throughput for all senders across all experiments. From the figure we find that CMAP improves the median per-sender throughput by a factor of 1.8 compared to 802.11.

We next study how the header or trailer reception probabilities at a node are affected by the number of concurrent transmissions in the network. Figure 19 shows the mean, median, and various percentile values of the probability of reception of either a header or trailer of a virtual packet at each receiver, as a function of the number of concurrent transmissions in the network. We find from the graph that the median header or trailer reception probability is practically unaffected by the number of concurrent transmissions. However, the 10th percentile value drops sharply, indicating that a small fraction of receivers cannot implement the conflict map mechanisms effectively in the presence of many concurrent transmissions. To increase the robustness of CMAP to header or trailer loss in such cases, we can replicate the header and trailer information (an overhead of 24 bytes) in every data packet of a virtual packet.

5.7 Mesh Topology

In this section, we present an evaluation of CMAP over a two-hop content dissemination mesh network shown in Figure 11(d). The source S first broadcasts a batch of packets to its one-hop neighbors A_1 , A_2 , and A_3 . The A_i s then transmit the packets to the corresponding B_i s. We compute the throughput at each B_i as the minimum of the throughputs along the corresponding $S \rightarrow A_i$ and $A_i \rightarrow B_i$ paths.

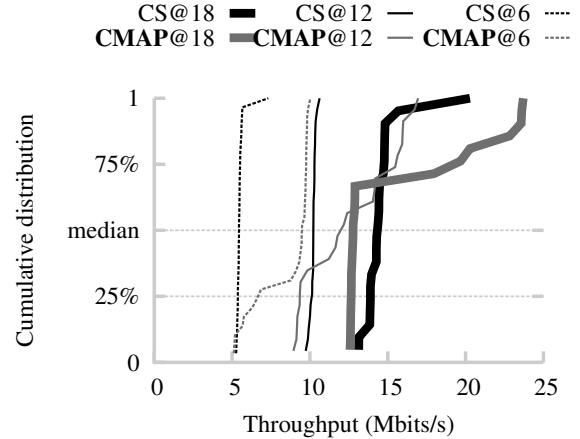


Figure 20. Experiment with exposed terminals transmitting at 6, 12 and 18 Mbits/s rates of 802.11a. CMAP continues to exploit exposed terminal opportunities at higher bit-rates.

We measured the aggregate throughput at all the B_i s over 10 different topologies. We found that CMAP achieves a 52% higher average throughput than 802.11 with carrier sense enabled. The reason for this improvement was that, frequently, one or more of the A_i s were exposed terminals during the $A_i \rightarrow B_i$ transfers.

While the above experiment may not be representative of the performance of CMAP over arbitrary mesh networks, it convinces us that multi-hop mesh networks can benefit from a MAC that exploits concurrent transmission opportunities. In fact, given a MAC like CMAP that increases concurrency, routing protocol design should be rethought to generate topologies that increase concurrent transmission opportunities.

5.8 Variable Bit-rates

Thus far all experiments have reported performance figures from the 802.11a 6 Mbits/s bit-rate. In practice, however, wireless senders may run at many of the higher bit-rates available to increase throughput. Consequently, we seek to measure if CMAP can continue to exploit exposed terminal opportunities at higher bit-rates.

We repeat the experiment with exposed terminals, as in §5.2, at the 12 and 18 Mbits/s rates of 802.11a. In each experiment, all the senders run at a common bit-rate that we set manually, and do not perform any online rate adaptation. Also, CMAP nodes were always made to transmit interferer list updates, headers, trailers and ACK packets at the lowest bit-rate irrespective of the bit-rate of data packets, resulting in a slight performance penalty at higher bit-rates. The CDFs of the throughputs of CMAP and 802.11 with carrier sense enabled at bit-rates of 6, 12 and 18 Mbits/s are shown in Figure 20. We see from the figure that CMAP

continues to show throughput improvements over 802.11 at higher bit-rates. Note however that the number of exposed terminal opportunities decreases with increasing bit-rates because the minimum SINR required to decode an incoming packet increases with increasing bit-rates; as a result, some pairs of links that could transmit concurrently at lower rates cannot do so at higher rates.

6 Related Work

Spatial reuse is a well-known concept in wireless communications networks of many different types. MACA [7] makes the observation that carrier sense cannot make correct transmission decisions because it does not consider channel conditions at the receiver, resulting in problems like exposed and hidden terminals. The paper proposes the RTS/CTS virtual carrier sensing protocol to solve the hidden terminal problem. However, this mechanism does not solve the exposed terminal problem.

In a busy access point WiFi network, Judd [6] observes that many clients connected to different access points are in fact exposed terminals with respect to each other. In fact, two randomly-chosen clients are *as likely* to be exposed terminals with respect to each other as they are to connect to the same access point. This result suggests that the use of conflict maps could significantly improve performance in infrastructure wireless LANs.

There have been a few previous proposals to increase concurrency in wireless networks [1, 11, 16, 3]. As we explain below, however, CMAP differs from all these schemes in the method of identifying and exploiting the identified concurrent transmission opportunities. Also, these previous proposals build upon the RTS/CTS mechanism and evaluate their ideas in simulation alone. Like CMAP, the “adaptive learning” extension of MACA-P [1] builds a data structure containing potentially non-interfering but nearby nodes. Unlike CMAP however, MACA-P is based on the RTS/CTS exchange, extended in time to include a *control gap*, which results in a significant protocol overhead. RTSS/CTSS [11] uses an offline training phase to determine which nodes may transmit concurrently. This approach, however, is not applicable when the channel varies, as is the case in practice. It also does not have any mechanisms to deal with the ACK loss problem in exposed terminals. Shukla et al. [16] propose identifying exposed terminals performing an RTS/CTS exchange on the basis of overhearing an RTS without overhearing a CTS. This method does not identify all exposed terminal opportunities—it misses exposed terminals where a sender can hear another receiver’s CTS, but is far enough from the receiver that it can transmit concurrently. In the Interference Aware (IA) MAC protocol [3], nodes make transmission decisions using the SINR estimates at receivers that are embedded in CTS messages. However, the IA MAC misses exposed terminals where one

of the exposed senders does not hear the CTS from the other receiver.

There have also been proposals to use receiver-based feedback of channel conditions in making transmission decisions to improve the performance of CSMA. E-CSMA [4] uses observed channel conditions at the transmitter (RSSI, for example), and receiver-based packet success feedback to build a per-receiver probability distribution of transmission success conditioned on the channel conditions at the sender at the time of transmission. Then a node makes a transmit/defer decision based on transmitter channel conditions just before sending a packet. The distinguishing feature of CMAP from E-CSMA is that CMAP explicitly takes the identity of current senders and whom they’re sending to into account while making channel access decisions, instead of implicitly capturing them using signal strength estimates, and hence can better predict which transmissions are likely to succeed and which not.

Other researchers have observed that concurrent transmissions do not always result in both the colliding packets being lost [18, 20]. This phenomenon, in which a receiver can correctly decode its sender’s packet even in the presence of other concurrent transmissions, is sometimes referred to as the “capture” effect. CMAP increases the opportunities for and exploits packet capture by increasing the number of concurrent transmissions. Whitehouse et al. [20] and Priyantha [15] propose mechanisms to boost the chances of packet capture. In these schemes, receivers acquire packet preambles throughout the duration of ongoing packet receptions in addition to when no packet is being received, thereby capturing stronger transmissions that start during the reception of weaker transmissions. These schemes can improve CMAP’s performance by helping it build up state about the network gleaned from stronger and later transmissions.

Padhye et al. [14] propose a set of metrics that estimate link interference in static multi-hop wireless networks. They suggest an offline process of pairwise link measurements to identify conflicting transmissions. Similarly, the interference map [13] builds up packet success information about CSMA transmissions in an offline training phase, for the purpose of network planning. CMAP, in contrast, obtains interference information online.

Algorithms to tune the carrier sense threshold or power level alone [12, 17, 19, 21, 22] and algorithms that tune both carrier sense threshold and transmit power [8] build on the basic carrier sense mechanism. Thus, these algorithms make a fundamental trade-off between preventing hidden-terminal collisions and permitting exposed-terminal spatial reuse, and don’t fully take advantage of the many exposed terminal opportunities present in real networks. CMAP explicitly discriminates between conflicting and non-conflicting transmissions, avoiding this tradeoff.

7 Conclusions

We presented the design, prototype implementation, and experimental evaluation of CMAP, a reactive channel access protocol that uses empirical observations of packet loss to build a distributed data structure of interfering concurrent transmissions (called the *conflict map*) in a wireless network. We showed how nodes can use this conflict map data structure to transmit concurrently with each other when transmissions do not interfere—almost all exposed terminal configurations see a $2\times$ gain in our experiments—while successfully avoiding conflicting concurrent transmissions, thereby increasing the aggregate throughput of the system. Though flows under CMAP may experience transient packet loss before conflict map entries converge, we believe that the gains with CMAP outweigh this loss, especially in topologies with scope for spatial reuse. Our evaluation of CMAP over realistic access point topologies with multiple concurrent senders shows that CMAP improves aggregate throughput by up to 47% and median per-sender throughput by $1.8\times$ over 802.11 by exploiting concurrent transmission opportunities.

Acknowledgments

We thank our shepherd Brad Karp and the anonymous reviewers for their insightful comments. This work was supported in part by the National Science Foundation under Grants CNS-0721702 and CNS-0520032, and by a Cisco Fellowship.

References

- [1] A. Acharya, A. Misra, and S. Bansal. Design and Analysis of a Cooperative Medium Access Scheme for Wireless Mesh Networks. In *Proc. of IEEE BROADNETS*, pages 621–631, San Jose, CA, Oct. 2004.
- [2] S. Biswas and R. Morris. Opportunistic Routing in Multi-Hop Wireless Networks. In *Proc. of ACM SIGCOMM*, pages 69–74, Philadelphia, PA, August 2005.
- [3] M. Cesana, D. Maniezzo, P. Bergamo, and M. Gerla. Interference Aware (IA) MAC: an Enhancement to IEEE 802.11b DCF. In *Proc. of IEEE Vehicular Technology Conf.*, number 5, pages 2799–2803, Oct. 2003.
- [4] S. Eisenmann and A. Campbell. E-CSMA: Supporting Enhanced CSMA Performance in Experimental Sensor Networks using Per-neighbor Transmission Probability Thresholds. In *Proc. of IEEE INFOCOM*, pages 1208–1216, Anchorage, AK, May 2007.
- [5] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *Proc. of ACM SIGCOMM*, pages 409–420, Kyoto, Japan, August 2007.
- [6] G. Judd. *Using Physical Layer Emulation to Understand and Improve Wireless networks*. PhD thesis, CMU, October 2006. CMU-CS-06-164.
- [7] P. Karn. MACA - A New Channel Access Method for Packet Radio. In *ARRL/CRRR Amateur Radio 9th Computer Networking Conf.*, September 1990.
- [8] T.-S. Kim, H. Lim, and J. Hou. Improving Spatial Reuse through Tuning Transmit Power Carrier Sense Threshold and Data Rate in Multihop Wireless Networks. In *Proc. of ACM MobiCom*, pages 366–377, Los Angeles, CA, Sept. 2006.
- [9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.
- [10] Multiband Atheros Driver for Wireless Fidelity. <http://madwifi.org>.
- [11] K. Mittal and E. Belding. RTSS/CTSS: Mitigation of Exposed Terminals in Static 802.11-Based Mesh Networks. In *Proc. of IEEE WiMesh Workshop*, Reston, VA, Sept. 2006.
- [12] J. Monks, V. Bharghavan, and W. Hwu. A Power Controlled Multiple Access Protocol for Wireless Packet Networks. In *Proc. of IEEE INFOCOM*, pages 219–228, Anchorage, AK, Apr. 2001.
- [13] D. Niculescu. Interference Map for 802.11 Networks. In *Proc. of the ACM/USENIX Internet Measurement Conf.*, San Diego, CA, Oct. 2007.
- [14] J. Padhye, S. Agarwal, V. Padmanabhan, L. Qiu, A. Rao, and B. Zill. Estimation of Link Interference in Static Multi-hop Wireless Networks. In *Proc. of the ACM/USENIX Internet Measurement Conf.*, Berkeley, CA, Oct. 2005.
- [15] N. B. Priyantha. *The Cricket Indoor Location System*. PhD thesis, MIT, June 2005.
- [16] D. Shukla, L. Chandran-Wadia, and S. Iyer. Mitigating the Exposed Node Problem in IEEE 802.11 Ad Hoc Networks. In *Proc. of IEEE ICCCN*, pages 157–162, Dallas, TX, Oct. 2003.
- [17] D. Son, B. Krishnamachari, and J. Heidemann. Experimental study of the effects of Transmission Power Control and Blacklisting in Wireless Sensor Networks. In *Proc. of the IEEE Conf. on Sensor and Ad-hoc Communication and Networks*, pages 289–298, Santa Clara, CA, October 2004.
- [18] D. Son, B. Krishnamachari, and J. Heidemann. Experimental Analysis of Concurrent Packet Transmissions in Low-Power Wireless Networks. In *Proc. of ACM SenSys*, pages 237–250, Boulder, CO, Nov. 2006.
- [19] R. Wattenhofer, L. Li, P. Bahl, and Y. Wang. Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad Hoc Networks. In *Proc. of IEEE INFOCOM*, pages 1388–1397, Anchorage, AK, Apr. 2001.
- [20] K. Whitehouse, A. Woo, F. Jiang, J. Polastre, and D. Culler. Exploiting the Capture Effect for Collision Detection and Recovery. In *IEEE EmNets Workshop*, Sydney, Australia, May 2005.
- [21] X. Yang and N. Vaidya. On Physical Carrier Sensing in Wireless Ad Hoc Networks. In *Proc. of IEEE INFOCOM*, number 4, pages 2525–2535, Miami, FL, Mar. 2005.
- [22] J. Zhu, X. Guo, L. L. Yang, W. S. Conner, S. Roy, and M. M. Hazra. Adapting Physical Carrier Sensing to Maximize Spatial Reuse in 802.11 Mesh Networks. *Wiley Journal of Wireless Communications and Mobile Computing*, 4(8):933–946, December 2004.

Notes

¹In the case of non-802.11 interference, CMAP cannot decode headers and hence does not defer transmissions as carrier sense with energy detect may. However, most 802.11 chipsets use preamble detection for carrier sense instead of energy detection.

²We are assuming that all sources transmit at the same power level always, independent of which destination they're transmitting to. We are also assuming omnidirectional, half-duplex radios.

³We disable link-layer ACKs to avoid the senders backing off in response to packet losses.

Designing High Performance Enterprise Wi-Fi Networks

Rohan Murty[†], Jitendra Padhye[‡], Ranveer Chandra[‡], Alec Wolman[‡], Brian Zill[‡]
[†]Harvard University, [‡]Microsoft Research

Use of mobile computing devices such as laptops, PDAs, and Wi-Fi enabled phones is increasing in the workplace. As the usage of corporate 802.11 wireless networks (WLANs) grows, network performance is becoming a significant concern. We have built DenseAP, a novel system for improving the performance of enterprise WLANs using a dense deployment of access points (APs). In sharp contrast with wired networks, one cannot increase the capacity of a WLAN by simply deploying more equipment (APs). To increase capacity, the APs must be assigned appropriate channels and the clients must make intelligent decisions about which AP to associate with. Furthermore, the decisions about channel assignment, and associations must be based on a global view of the entire WLAN, rather than the local viewpoint of an individual client or AP. Given the diversity of Wi-Fi devices in use today, another constraint on the design of DenseAP is that it must not require any modification to Wi-Fi clients. In this paper, we show how the DenseAP system addresses these challenges, and provides significant improvements in performance over existing WLANs.

1 Introduction

In a typical office environment, the wired network is generally well-engineered and over-provisioned [12]. In contrast, deploying 802.11 wireless networks (WLANs) in enterprise environments remains a challenging and poorly understood problem. WLAN installers typically focus on ensuring coverage from all locations in the workplace, rather than the more difficult to measure properties such as capacity or quality of service. Thus, WLAN users commonly experience significant performance and reliability problems.

The usage model for enterprise WLANs is currently undergoing a significant transformation as the “culture of mobility” takes root. Many employees now prefer to use their laptops as their primary computing platform, both in conference rooms and offices [18]. A plethora of handheld Wi-Fi enabled devices, such as PDAs, cell phones, VoIP-over-Wi-Fi phones, and personal multime-

dia devices are becoming increasingly popular. In addition to the scalability challenges that arise with increased WLAN usage, the applications for many of these new mobile devices require better QoS and mobility support.

The need to improve enterprise WLAN performance has been recognized by both the research community [20, 19, 7, 10, 24] as well as industry [3, 5, 2, 4, 1]. Upgrades at the PHY layer, such as the transition from 802.11g to 802.11n, are important steps along the path to increasing WLAN capacity, but they are not enough. Deploying more APs has the potential to improve WLAN capacity, but one must also address issues such as channel assignment, power management, and managing association decisions.

In this paper, we present a new software architecture called DenseAP, that supports a dense deployment of APs to significantly improve the performance of corporate WLANs. A key emphasis in our design of the DenseAP system is on practical deployability. Because of the incredibly wide diversity of existing Wi-Fi devices, DenseAP must provide significant performance benefits without requiring any modifications to existing Wi-Fi clients. Furthermore, we do not consider any changes that require hardware modifications or changes to the 802.11 standard.

The DenseAP architecture and design challenge two fundamental characteristics of most current enterprise WLAN deployments. First, existing WLANs are designed with the assumption that there are far fewer APs than clients active in the network. In the DenseAP architecture however, the APs are deployed densely – in the common case there may be an AP in every office. Second, in conventional WLANs clients decide which AP to associate with, whereas the DenseAP system uses a centralized association control.

The scarcity of APs in conventional enterprise WLANs limits their performance in a variety of ways. For example, with a large number of non-overlapping channels (e.g. 12 in 802.11a) but only a few APs, the

WLAN is unable to fully utilize the available spectrum at each location. Because radio signals fade rapidly in indoor environments, adding extra radios to existing APs is not as effective as deploying a larger number of APs in different locations. If APs are densely deployed, each client can associate with a nearby AP, and will see better performance. A dense deployment also reduces the impact of the “rate anomaly” problem [13] that hurts the performance of conventional WLANs.

With a dense deployment of APs, clients have many possible APs to choose from, and therefore access point selection policy is critical to achieving good performance. In conventional WLANs, clients select which AP to associate with using only locally available information. Most clients use signal strength as the dominant factor in selecting an AP, yet it is well-known that this behavior can lead to poor performance [14]. For example, when many clients congregate in a conference room, they all tend to select the same AP even when multiple APs operating on different channels are available. To improve performance in this scenario, clients must associate with different APs.

In the DenseAP architecture, a central controller gathers information from all APs, and then determines which AP each client should associate with. Simultaneously, the central controller also decides on the assignment of channels to APs. Even though Wi-Fi clients implement their own association policies and we do not modify these clients, the DenseAP controller effectively bypasses the client association policy by only exposing to each client the particular AP with which it wants the client to associate. Using a similar technique, the DenseAP controller also carries out periodic load balancing by seamlessly moving clients from overloaded APs to nearby APs with significantly less load.

The DenseAP architecture is quite versatile, and capable of improving many aspects of performance of enterprise WLANs. In this paper, we primarily focus on describing how DenseAP significantly improves the capacity of enterprise WLANs. We define capacity simply as the sum total of throughput all active clients in the network can potentially achieve. We will also briefly discuss how the architecture impacts other aspects of performance, such as quality of service for delay and jitter sensitive applications.

One obvious question that arises when considering a dense AP deployment is whether the performance gains justify the costs. One approach to reducing equipment costs is to leverage existing enterprise desktops and convert them to APs, similar to our previous work on DAIR [8]. However, the key concern of enterprise IT departments when deploying any new technology is typically the people costs associated with managing that technology. The DenseAP system is designed to require

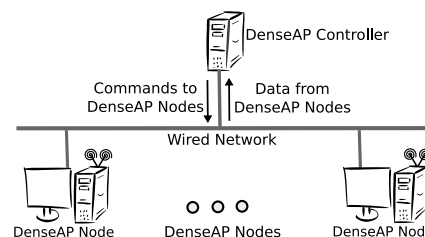


Figure 1. Overall architecture of the DenseAP system.

very little management overhead: the DenseAP nodes are self-configuring, and the redundancy available from the dense deployment means that AP hardware failures do not need to be addressed immediately.

This paper makes the following new contributions. First, our system supports a high density of APs with off-the-shelf, completely unmodified clients. As a result, it provides performance benefits for all clients, including the *many* different types of handheld Wi-Fi devices that have recently appeared. Second, we demonstrate the performance benefits of our system at a significantly higher density than previous work. Third, we demonstrate that intelligent management of the association process is necessary even when you have a very high density installation of APs. Forth, we present a novel load estimation technique that allows our system to automatically factor in impact of external interference, such as traffic from nearby networks.

We have deployed the DenseAP system with 24 APs in our offices. The testbed can function in both 802.11a and 802.11g modes. Our experiments show that the system provides large improvements in performance over the existing corporate network. In specific cases, the improvement in throughput can be as large as 1250%. We present a series of experiments that show how various aspects of our system work together to provide these gains. We also show that our system is capable of handling nomadic and mobile clients.

2 Design Overview

Figure 1 is a high-level illustration of the DenseAP system. The system consists of several DenseAP nodes (DAPs) which provide wireless service and a DenseAP controller (DC) which manages the DAPs.

A DAP is a programmable Wi-Fi AP connected to the wired network. Each DAP periodically sends summaries to the DenseAP controller comprising of a list of associated clients, their traffic pattern summaries, RSSI values of a few packet samples from their transmissions, current channel conditions, and reports of new clients requesting service from the network. We classify DAPs into two categories: we refer to DAPs that do not have any clients associated with them as *passive*; those that have at least

one associated client are called *active*.

The DC manages the DAPs. The periodic reports sent by the DAPs provide the DC with a global view of the network activity. Using this global view, the DC selects the right DAP for a client, allocates channels to DAPs, performs load balancing when needed, handles client mobility, and deals with DAP failures.

In the next section, we describe the mechanisms the DC uses to ensure that the client associates with the selected DAP. Then, we describe the algorithms involved in selecting the DAP and a channel for the DAP. We discuss power control and related issues in Section 5

3 Association Mechanism

In conventional WLANs, APs advertise their presence by sending out Beacon frames which include their SSID and BSSID. Prior to association, clients gather information about the APs by scanning the channels one by one, and listening for Beacons on each channel. This is called “passive scanning”. The clients also perform “active scanning”, whereby they send out a Probe Request frames on each channel. These are requests for APs to send out information about themselves. APs respond to Probe Requests with Probe Response frames, the contents of which are similar to Beacon frames. Once the client gathers information about all APs, it decides which AP to associate with.

The 802.11 standard allows APs to beacon with the SSID field set to null – this is referred to as a hidden SSID. A client that wishes to associate with an AP using a hidden SSID must first send out a Probe Request that contains the SSID of that network, which will then cause the AP to provide a Probe Response. For any client that does not provide the correct SSID, the AP does not respond.

The DC performs association control by exposing DAPs to clients on a “need to know basis”. This is achieved as follows. First, the passive DAPs (i.e., those that do not have clients associated with them) in the network do not send out any beacons. The active DAPs do send out beacons but with a hidden SSID. Second, each DAP maintains a local access control list (ACL) of client MAC addresses. On receiving a probe request from a client, the DAP replies with a probe response message only if the client’s MAC address is in its ACL. If a DAP receives a probe request (it may be a broadcast request) from a client whose MAC address is not in its ACL, it sends a message to the DC informing the controller that a client might be requesting service. The DC determines which, if any, DAP should respond to the probe request and adds the MAC address of the client to the ACL of that DAP.

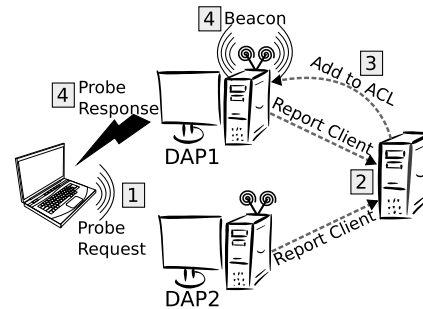


Figure 2. Association in the DenseAP system

By adding the MAC address of a client to only one DAP’s ACL at a time, the DC ensures that for the SSID associated with the DenseAP network, *only one* DAP is visible to the client at any given time.

Note that traditional MAC address filtering could not have achieved this. MAC address filtering only prevents association, not probe responses. With traditional MAC address filtering, a client would discover several DAPs, and it may not even try to associate with the one that the DC has chosen for it.

We will now illustrate how these two techniques are used when a client associates with the system for the first time, and handing off a client from one DAP to another.

3.1 Associating New Clients

Figure 2 illustrates the association mechanism instrumented in DenseAP. (i) A new client C with mac address C^{mac} broadcasts probe requests. (ii) DAPs DAP_1 and DAP_2 receive probe requests and inform the DC of C^{mac} . (iii) The DC executes the association algorithm and determines which AP the client should associate with. Assuming it picked DAP_1 , the DC then sends a message to DAP_1 to add C^{mac} to its access control list (ACL_{DAP_1}). (iv) DAP_1 , on receiving the next probe request from C , checks to see if $C^{mac} \in ACL_{DAP_1}$. If so, it responds to C with a probe response thus initiating the association process. If DAP_1 was not previously beaconing, it now begins.

The reader may wonder why DAPs beacon at all. One reason is that beacons are essential for allowing the clients to enter power-save mode. In addition, certain popular drivers automatically disconnect from an AP if they do not receive periodic beacons.

We note a few points about this mechanism. First, we have verified that this mechanism works with the device drivers of a variety of popular 802.11 chipsets including Atheros, Intel Centrino, Realtek, Ralink, and Prism2. Second, if a client fails to associate with the assigned DAP (e.g. due to interference near the client), the DC detects this since DAPs periodically report back information about their associated clients. The DC then re-assigns the client to a different DAP. Third, ACL entry for a client is maintained only as long as the client is as-

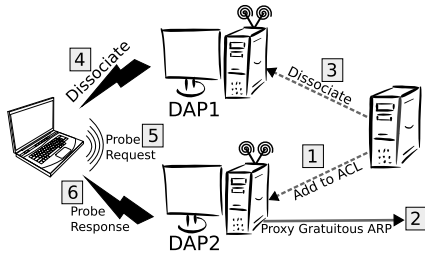


Figure 3. An example handoff in the DenseAP system. The client is switched from DAP_1 to DAP_2 .

sociated with the AP.

3.2 Client Handoffs

The DC may sometimes want to handoff clients from one DAP to another, for load balancing or because the client's location has changed. Figure 3 illustrates the sequence of steps that lead to a seamless handoff in DenseAP.

Assume that client C has successfully associated with DAP_1 . The following steps are taken when the DC decides to handoff C from DAP_1 to DAP_2 . (i) The DC adds C_{mac} to the ACL on DAP_2 . (ii) To ensure any further traffic flowing towards C is routed via DAP_2 , DAP_2 sends out a gratuitous proxy ARP message containing C_{IP} and DAP_2^{mac} to the wired subnet. (iii) The DC asks DAP_1 to send a *disassociate* frame to C . (iv) DAP_1 removes C^{mac} from the local ACL and sends a disassociate frame to C . It also cleans up all local association state pertaining to C . (v) C receives the disassociate frame and immediately begins to scan for other DAPs by sending out probe requests. (vi) Upon receiving C 's probe request, DAP_2 responds with a probe response.

After associating with DAP_2 , C does not send out a DHCP request since the time taken to re-associate did not cause a local *media-disconnect* event. Therefore, it is important to ensure that the DC only hands off clients between DAPs that are on the same subnet. In practice, we expect that all DAPs managed by a DC will be on the same wired subnet. We present results illustrating the efficacy of our handoff mechanism in Section 6, along with time required for each step.

Having described the mechanisms by for enforcing the association decisions, we now describe the policy for making these decisions.

4 Association Policy

The goal of the association policy is to improve the overall system capacity. We do this by picking the “right” DAP for a client to associate with, and when needed we select the “right” channel for that DAP to operate on.

Intuitively, the way to improve overall system capacity is to have each client to associate with a nearby,

lightly-loaded AP. Furthermore, APs that are close to each other should operate on orthogonal channels. We will formalize these notions in the rest of the section.

We do not claim that our association policy is optimal. Rather, our aim is to provide a significant improvement over existing WLAN networks, by taking advantage of the dense deployment of DAPs, and without requiring any changes to the clients.

In this section, we first present a metric we call *Available Capacity* to rank all possible DAPs that a client can associate with. We then describe the association policy for four scenarios: (i) when a new client shows up, (ii) when the wireless channel conditions change, (iii) when clients move, and (iv) when DAPs fail. A more detailed description of our association policy and related issues is available in [22].

4.1 The Available Capacity Metric

When we select a DAP for a client to associate with, we want to pick a DAP where a client can expect to get good throughput. Unfortunately, it is not easy to determine what throughput a client can expect to get when associated with a DAP. The value depends on several factors, such as quality of the channel between the DAP and the client, presence of other traffic/interference, autorate algorithms in use and the CCA thresholds used by the client and the AP. Rather than estimating each of these factors, we focus on the two that affect the expected throughput the most, namely: the transmission rate the client and the DAP can use to communicate with each other; and how busy the wireless medium is in the vicinity of the client and the AP.

The transmission rate is a function of, among other things, how well the signal propagates between the client and the DAP. For example, if a DAP and a client are far away from each other, they generally won't be able to communicate at high transmission rates, since the wireless signal degrades with distance. Alternatively, when a DAP and a client are close to each other, they will be able to communicate at higher transmission rates. However, if the wireless channel is busy with other traffic, the expected throughput will be lower, since the client and the DAP will have fewer opportunities to transmit packets.

The combined impact of the transmission rate and the busy medium is approximated by the *Available Capacity* (AC) metric as follows. Given a channel (C), a DAP (D) and a client (M), AC_{DM}^C is the product of *free air time* on C in the vicinity of D and M , and the *expected transmission rate* between the D and M . The free air time is simply the percentage of time when the wireless medium is not in use. Our notion of load at a DAP on a particular channel is defined by $(1 - \text{free air time})$.

The intuition behind this metric is that the DAP with the highest available capacity will allow the client to send

the most data, while simultaneously reducing the impact on other clients in the network. For example, if a client and DAP can communicate at a high transmission rate, then each frame will consume less air time, and the client will be able to send more data. Furthermore, other clients on the same channel in the vicinity will have more opportunity to transmit. On the other hand, if the channel has low utilization then it is acceptable for the client and the DAP to communicate at a low transmission rate because it will have little impact on other clients.

This metric is similar to the one proposed in [10], where clients associate with the AP which is the least loaded and offers the best data rate. We compare and contrast our work with [10] in more detail in Section 8.

We now describe how we estimate the free air time and the expected data rate for a given DAP/client pair. We stress that we do not expect these calculations to be precise. Our intention is to provide a reasonable ordering of DAPs, in order to pick a good AP for the client to associate with. The load balancing process described later in the section can reassign the client to a different DAP, should the conditions change and the initial choice is no longer appropriate.

4.1.1 Estimating Free Air Time

Given a dense deployment of DAPs, it is likely that the DC will associate a client with a nearby DAP. Hence, it is also likely that the wireless channel conditions near the DAP and the client associated with it will be similar. Thus, we only estimate the free air time on a channel in the vicinity of the DAP.

The amount of free air time around the DAP depends on the traffic generated by the DAP itself (i.e. downlink traffic), the uplink traffic and any background traffic/interference. The background traffic includes traffic generated by other DAPs and clients in the vicinity that are functioning on the same channel. Of these quantities, the DAP can easily determine the amount of air time consumed by the traffic it generates. To determine the rest, we have devised a method inspired by the ProbeGap technique [16].

Each DAP periodically sends a small broadcast packet at a fixed transmission rate on the highest priority driver queue, which is usually reserved solely for high priority PSM packets. The packets in this queue are sent even if the DAP has other packets pending in the normal data queue.

The DAP records the difference in time from when the packet was queued to when the transmit completion interrupt signals that the frame was successfully sent. When the channel is idle, the packet will be transmitted immediately and the measured delay will be the sum of frame transmission time plus various OS overheads. We denote this value by δ_{min} . Because the channel is idle,

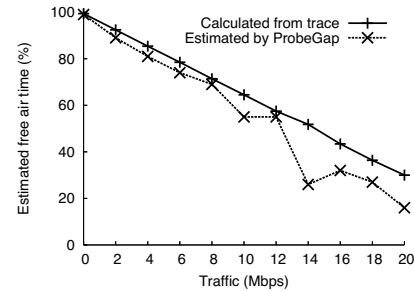


Figure 4. Validation of ProbeGap method

δ_{min} does not depend on channel conditions. The transmission time of the frame is determined by the transmission rate and the size of the packet, both of which are constant. The OS overheads depend on the hardware and software configuration of the DAP. If the channel is busy, the packet will be delayed by more than δ_{min} . This additional delay depends on several factors such as number of contending stations, the size of packets they are sending and their transmission rates. To obtain a good estimate of the fraction of time the channel is busy, we send a number of such probe packets and count the fraction of packets that experienced a delay of more than δ_{min} . Because δ_{min} does not depend on channel conditions, it can be calculated apriori by performing simple calibration experiments.

This technique works quite well in practice, as shown in Figure 4. The experiment was performed as follows. We set up a DAP with no associated clients on an otherwise idle channel in the 802.11a band. We had previously determined that δ_{min} was 250 microseconds. Using another AP and a client, we generated different volumes of CBR UDP traffic on the channel. We had also set up a sniffer to capture every packet. This graph shows the amount of free air time estimated by the DAP, as well as the “actual” free air time calculated using the trace captured by the sniffer. We see that the estimate provided by our method is a good approximation, although we tend to slightly underestimate the free air time.

Note that the probe packets are not delayed by the normal data traffic generated by the DAP itself (i.e. downlink traffic). So, to estimate the free air time, the DAP adds the air time consumed by the traffic it generates to the estimate obtained by the ProbeGap method before reporting it to the DC.

If a DAP has clients associated with it, it only needs to report the free air time for the channel it is currently on. If a DAP has no associated clients, it can scan all channels and report to the DC the channel that has the most free air time.

In our current implementation, each DAP sends the load estimation probes every 200ms. The free air time is estimated over a period of 20 seconds.

4.1.2 Estimating Expected Transmission Rate

It is difficult to accurately predict the transmission rate a client will achieve when communicating with a DAP (or vice-versa). The rate primarily depends on how well the DAP receives the client's signal. However, the rate also depends on a variety of other factors such as the autorate algorithm implemented by the client, power levels used by the client, and channel conditions near the client. Of these factors, we can only estimate how well the DAP receives a client's signal.

When attempting to associate, clients send out probe request messages which are overheard by nearby DAPs who then inform the central controller. We estimate the quality of the connection between the client and the various candidate DAPs using the signal strength (RSSI) of the received probe request frames at the various DAPs. We convert these observed signal strengths into estimates of expected transmission rate by using a mapping table. The mapping table bucketizes RSSI values into fixed-size buckets, and assigns an expected rate to each bucket. We assume that the same transmission rate will be used by both the client and the AP. We call this the *rate-map* approach. The mapping table is initially generated by manual profiling using a few clients at various locations. It can then be refined as actual data from more clients is gathered during live operation.

At first glance, it may appear that extrapolating the signal strength observed in the uplink direction to an expected transmission rate in both directions could result in inaccurate estimations and/or poor performance, especially considering the other factors that are ignored. Yet, in our system, we find that it provides reasonable results for the following reasons. First, given the density of access points, a client generally associates with a nearby DAP. For such short distances, we find that signal strength measured in one direction is a good approximation of signal strength seen in the other direction. Second, because the client and the DAP are usually close to each other, we generally see good signal strength in both directions. Most commercial Wi-Fi cards behave similarly in such conditions. Finally, note that we do not need the exact transmission rates used by either the client or the DAP. The conversion table is merely a way to ranking the relative importance of the observed signal strength. In Section 6, we will present results that demonstrate the usefulness of the rate-map approach.

We now describe how the AP selection algorithm uses the available capacity metric.

4.2 Associating a New Client

When a new client first appears in the network, it scans on all channels and sends out probe requests. Because this client has not yet been added to the ACL of any

DAP, all DAPs that hear the probe requests simply report them to the DC. To calculate reasonable signal strength estimates, the DC waits for a short while (10-30 seconds in our current implementation) after the first report of a new client is received. During this interval it continues to collect reports of probe request packets from DAPs. At the end of this interval the DC calculates the average signal strength of all the probe request frames seen by each DAP. The rest of association algorithm is illustrated by the following example.

Assume two DAPs, A and B hear probe requests from a client M . Assume that A is active i.e. it already has other clients associated with it, whereas B does not (passive). For both A and B , the DC first calculates the expected rates with M , R_{AM} and R_{BM} , using both the observed signal strengths and the rate map table. Then, the DC considers the amount of free air time at each DAP. A already has clients associated with it, and therefore it is operating on some channel X . Hence, A has already been reporting free air time for that channel. We denote this by F_{XA} . Using the most recent report, the DC calculates the available capacity at A on channel X by $AC_{AM}^X = F_{XA} * R_{AM}$.

Now let us consider DAP B . It has no clients associated with it. Let us assume that DAP B has recently seen the highest available free air time on channel Y , denoted by F_{YB} . The DC calculates the available capacity at B on channel Y as $AC_{BM}^Y = F_{YB} * R_{BM}$. The DC then compares AC_{AM}^X and AC_{BM}^Y , and picks the higher of the two. If they are equal, it decides in favor of B , since B has no clients associated with it. In general, whenever available capacity of several DAPs are equal, the DC always picks one that has the fewest clients associated with it. If the DC picks A , it adds M 's mac address to A 's ACL. If it picks B instead, it first instructs B to stop scanning and to stay on channel Y . It then adds M 's mac address to B 's ACL. In both cases, the rest of the association process unfolds as described in Section 3.1.

Note two key aspects of this algorithm. First, we never move existing clients to another DAP as a result of a new client association. Second, DAPs are only assigned channels on an *on-demand* basis, as part of the association process. A DAP is assigned a channel only when a client in its vicinity requests service from the network. When a DAP becomes passive, it no longer has an assigned channel.

4.3 Load Balancing

The goal of the load balancing algorithm is to detect and correct overload situations in the network. We expect that such situations will be rare in an environment with a dense deployment of access points, and with numerous available orthogonal channels (e.g. 12 in 802.11a). However, it is important to watch for, and correct the overload

situations if and when they occur.

For example, an overload situation might occur if many clients congregate in a conference room, and the network conditions are such that the algorithm described in Section 4.2 assigns several of them to a single DAP. In such a situation, all clients simultaneously transmitting or receiving data can cause an overload at the DAP.

The load balancing algorithm works as follows. Once every minute, the DC checks all DAPs to see if any are severely overloaded. Recall from Section 4.1.1 that the busy air time (load) calculation incorporates the impact of traffic/interference near the DAP and the downlink traffic generated by the DAP. We consider a DAP to be overloaded, if it has at least one client associated with it, and it reports free air time of less than 20%. In other words, the channel is more than 80% busy in the vicinity of this DAP. The DC considers the DAPs in the decreasing order of load. If an overloaded DAP (A) is found, the DC considers the clients of A as potential candidates to move to another DAP. Recall that the DAPs send periodic summaries of client traffic to the DC. These summaries include, for each client, a smoothed average of the sum of uplink and downlink traffic load generated by the client during the previous interval. The load is reported in terms of air time consumed by the traffic of this client, and the average transmission rate of the traffic.

For each client $M \in A$, the DC attempts to find a DAP B such that the expected rate M will get at B is no less than the average transmission rate of the client at A , and the free air time at B is at least 25% more than the air time consumed by M at A . If such a DAP is found, M is moved to B using the process described in Section 3.2. Note that if B had no clients associated with it, the DC will also assign it a channel (the one B reported to have the most free air time on), just as it would do when associating a new client.

The load balancing algorithm moves at most one client that satisfies the above criteria during each iteration. Furthermore, once a client M has been handed off from A to B , it is considered ineligible to participate in the next round of load balancing. These hysteresis techniques are intended to prevent oscillations.

We note a few things about the load balancing algorithm. (i) Our algorithm is conservative. Moving clients from one AP to another is a potentially disruptive event, and we try to minimize how often we force such reassociations to occur. (ii) The load balancing algorithm improves overall system throughput in two ways. First, the client that is moved to the less-loaded AP can ramp up and consume more bandwidth. Second, the clients that stayed with the previously overloaded AP now have one less client to contend with, and they can also increase their throughput. (iii) It is sometimes possible to do load balancing by changing the channel of the over-

loaded DAP. This technique is useful only if the background traffic/interference (potentially from other DAPs) on the channel is significantly higher compared to the traffic sent/received by the overloaded DAP itself. However, the drawback of this technique is that all clients associated with the DAP will have to re-associate. Since we consider client re-associations to be disruptive events, we do not use this technique.

4.4 Mobility

The DC keeps track of a client's location, using the algorithm described in [11]. The algorithm takes into account the signal strength of a client's transmissions as reported by various DAPs, and the location of those DAPs, to determine the approximate location of the client. The median location error is about 1.5 meters. This is sufficient for our purpose because we only need to detect that the client's location has changed significantly.

The DC updates the locations of clients in the system every 30 seconds. When a client's location changes by more than 10 meters, the DC finds another DAP for the client to associate with, using the criteria described in the previous section. If such a DAP is found, the client is handed off to the new DAP. A client that undergoes handoff is considered ineligible to participate in the subsequent round for load balancing to prevent oscillations. It is, however, eligible to participate in another, mobility-related handoff.

4.5 Fault Tolerance

DAPs send periodic reports to the DC, so it is easy for the DC to detect when a DAP fails. In our current implementation, if the DC does not receive any reports from a DAP for up to one minute, it flags the DAP as a possible failure and does not assign any new clients to it. The clients associated with the failed DAP get disconnected. These clients immediately begin scanning for other DAPs in the vicinity by sending out probe request messages. Other DAPs in the vicinity pick up these probe messages and alert the central controller, which assigns these clients to other DAPs, as per the association policy.

5 Power Control

In a dense deployment of DAPs, transmit power control can mitigate the effects of interference between DAPs on the same channel [20], and increase spatial reuse. Since we do not wish to modify clients, we must do power control at DAPs alone. However, such unilateral power level can cause the clients and APs to operate at different transmit power levels. Prior work [20] has shown that

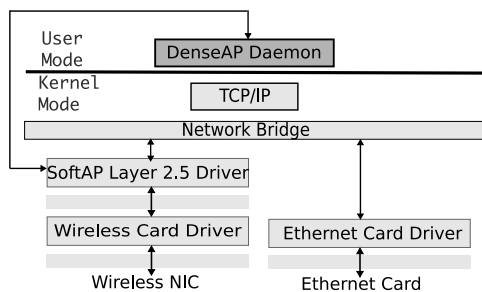


Figure 5. The network stack on each DenseAP node

asymmetric transmit power levels can increase the number of hidden terminals in a WLAN coverage area.

We have implemented and tested several adaptive power management schemes in our testbed. We do not present detailed results due to lack of space. Briefly, our results confirm the observations in [20]. Based on these results, as well as those reported in prior work, we conclude that unilateral power control at DAPs is undesirable, and the best policy is to simply use the maximum power level. A secondary benefit of this scheme is that it provides better coverage in the intended coverage area.

Two other parameters that can also affect the overall WLAN capacity are the Clear Channel Assessment (CCA) threshold used by each DAP [19], and the autorate algorithm implemented on each DAP. The wireless cards we used in our testbed do not allow us to change the CCA threshold. Auto-rating algorithms have been studied extensively by prior research. DenseAP nodes use the autorate algorithm described in [25].

6 Evaluation

We have built a prototype implementation of the DenseAP system. Figure 5 illustrates the network stack on each DAP. The network stack enables AP functionality on ordinary desktop machines. An integral part of the stack is our software AP (SoftAP), a fully programmable AP for the Windows platform. The wired and wireless interfaces are bridged. Each DAP also runs a DenseAP daemon, a user-level service responsible for managing local access point functionality. The service periodically queries the SoftAP driver, and sends summaries of client statistics to the DC. It also receives commands from the DC and sets appropriate parameters in the driver.

The DAPs are off-the-shelf PCs running Windows Vista, and the networking stack described above. Each machine is equipped with a Netgear JWAG511 wireless NIC. The wireless NICs are based on a chipset from RealTek. They support operation in 802.11 a/b/g modes, with one limitation: in the 802.11a mode, they can operate only on the lower 8 channels (channels 36-64). We also found that the RealTek cards do not work reliably

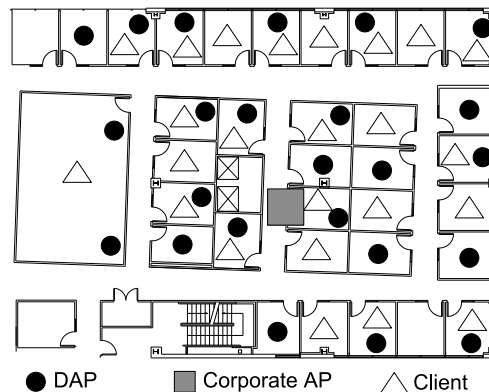


Figure 6. The testbed. The area is roughly 32m x 35m. The rooms have full walls, and solid wood doors.

in promiscuous mode, so we use an additional radio on each DAP to simulate promiscuous mode. This second radio is not fundamental to our approach, and is used only to compensate for the shortcomings of the RealTek card. All of the DAPs are connected to the same IP subnet on their wired Ethernet link. The DC is an ordinary desktop-class machine.

The DenseAP testbed is deployed on a portion of our office floor, as shown in Figure 6. The testbed consists of 24 DAPs. The DAPs are deployed roughly in every other office. Within each office, the machine is placed on the floor; the exact location determined by the consent of the occupant. This area of our building is served by a single corporate WLAN AP. The AP is located roughly at the center of the area, and is placed on the ceiling. Note that the DenseAP deployment is 24 times denser than the corporate WLAN. In addition to the DAPs, we have also deployed 24 machines to serve as clients. The clients are a mix of ordinary desktop and laptop machines, equipped with a variety of off-the-shelf wireless NICs.

Most of the the experiments reported in this paper were run in 802.11a mode (5 GHz band). There is very little corporate traffic in the 802.11a band. Thus, for most experiments, the background traffic is negligible, and we did not need to run the load estimation algorithm. The 802.11g band does see a fair amount of usage during normal office hours, but to avoid impacting corporate traffic, we were limited to conducting 802.11g experiments outside of normal work hours. We now turn to evaluating the performance of the DenseAP system using this testbed. We begin by validating the rate-map approach (section 4.1.2), which lies at the heart of our association and load balancing algorithms.

6.1 Usefulness of Rate-Map Approach

The rate-map approach (section 4.1.2) is the foundation of the association and load balancing algorithms. This approach is based on the hypothesis that the sig-

nal strength of a client's probe request packets, as observed by a DAP (i.e. *uplink* packets), is a good approximation of the transmission rate a client can expect in *both the up link and downlink* directions in a dense DAP deployment. Higher transmission rates, generally imply higher throughput between the corresponding DAP and the client. Hence, the objective of the rate-map approach is to pick a DAP such that a client will get good throughput in both directions.

To validate our hypothesis we demonstrate a positive correlation between the RSSI of the probe request packets from the client to both uplink and downlink throughput via the following experiment. We set up a client laptop at a fixed location. The client attempts to associate with each of the 24 DAPs in turn. Prior to each association attempt with a DAP, we measure the signal strength of the client's probe request packets as observed at that node. This is the uplink signal strength (USS). After associating with a DAP, the client contacts a server on the wired network, and carries out a 2 minute TCP download, followed by a 2 minute TCP upload. We carry out this experiment from 6 different locations, and repeated the entire process 5 times. The experiment was performed on channel 64 of 802.11a band.

We found a correlation of 0.71 between USS and upload throughput and 0.61 between USS and the download throughput. These results indicate that USS can be indeed be used as a good predictor of upload and download throughput. This correlation is much stronger if we look at the throughput numbers against bucketized USS values. The rate-map approach bucketizes USS values and assigns a rate to each bucket (section 4.1.2). Figure 7 illustrates the strong correlation between throughput numbers and these bucketized USS values. The error bars indicate one standard deviation. It can be seen that the bucketized USS values are a good predictor of both upload and download throughput.

We have conducted these measurements over a number of clients and consistently found positive correlations thereby validating the hypothesis that USS values of probe packets from clients, can be used as good proxies for transmission rates between a DAP and the client.

Note that we have demonstrated a correlation between USS and uplink and downlink *throughput*. Detailed results that demonstrate a correlation between USS and *transmission rates* are available in [22].

6.2 DenseAP Performance

We now present results that demonstrate the performance of the DenseAP system. We first establish the baseline for all our experiments. We then demonstrate the overall gains achieved by DenseAP and present a series of experiments that delineate the contribution of various facets of the DenseAP system to those gains. We

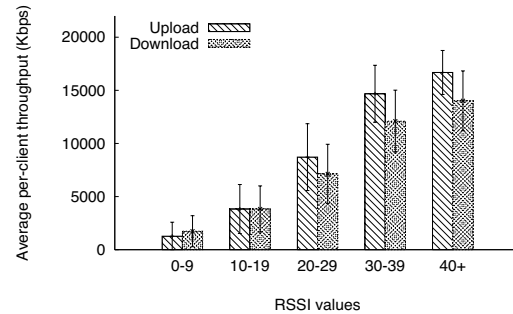


Figure 7. Correlation between bucketized uplink RSSI and upload/download throughput

also demonstrate graceful degradation of the system's performance when the number of clients increases or when number of DAPs decrease. Since a large majority of the traffic in WLANs tends to be downlink [11], we only present downlink numbers for most experiments. We have conducted uplink experiments in each case, and found the uplink results to be very similar to the downlink results.

6.2.1 Establishing the Baseline

We begin by evaluating the performance of the corporate WLAN, to provide a baseline against which we can compare the performance of our system.

As mentioned earlier, the testbed area is served by a single corporate AP. To establish the baseline performance, we had a group of clients associate with the corporate AP. The clients then simultaneously carried out a one minute TCP download from a server on the wired network. We varied the group size (the number of clients) from 2 to 12. The experiment is repeated 10 times for each group size. Each time, the group members are selected at random from among the available clients. We performed similar experiments for upload.

The results of this experiment are shown in Figures 8 and 9. Each point represents the median per-client throughput, and the error bars show SIQR.

For both 802.11a and 802.11g, the median per-client throughput drops as number of simultaneously active clients increases. However, the 802.11g numbers are substantially lower than the 802.11a numbers. This is because in 802.11g mode, the corporate AP sends out a CTS-to-self before every packet to avoid interfering with 802.11b clients. This is a well-known, and well-studied issue. Since we have not implemented the CTS-to-self feature for DAPs, we will refrain from directly comparing the performance of DenseAP and the corporate network in 802.11g mode.

6.2.2 Overall DenseAP Performance

We now repeat the experiment described in the previous section, but using the DenseAP system instead of

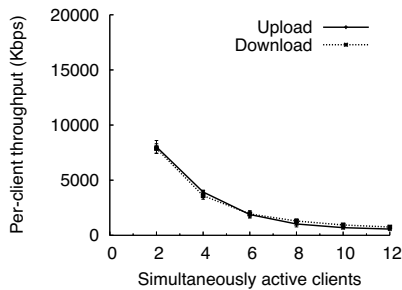


Figure 8. Baseline performance: 802.11a

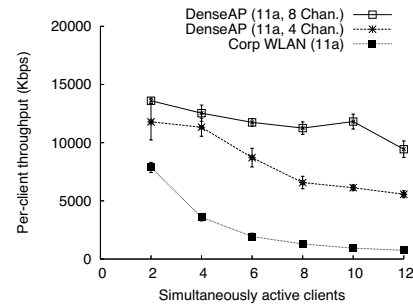


Figure 10. DenseAP performance: 802.11a

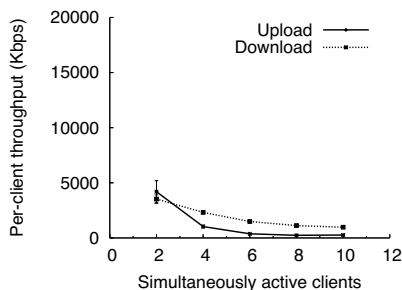


Figure 9. Baseline performance: 802.11g

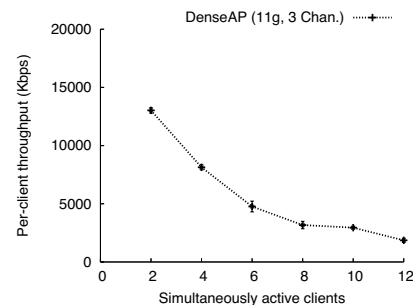


Figure 11. DenseAP performance: 802.11g

corporate WLAN. All features such as channel assignment, association policies and load balancing were enabled. We repeated the experiment twice in 802.11a mode, once using 8 channels (channels 36-64) and once using 4 channels (channels 40, 48, 56 and 64). We also ran the experiment with the DenseAP system in 802.11g mode using 3 orthogonal channels (channels 1, 6 and 11).

Figure 10 illustrates the performance of DenseAP in the 802.11a band. Figure 11 illustrates the performance of DenseAP in the 802.11g band. The graph does not have a baseline, since we do not wish to compare performance of corporate WLAN and DenseAP in 802.11g mode, as explained earlier. Let us focus on the 802.11a results.

We see significant performance gains over the corporate network. For example, with 8 simultaneously active clients, the median download throughput on the corporate network was 1.3Mbps. On the other hand, the median download throughput with DenseAP when using 8 channels, was 11.25Mbps. This represents an improvement in capacity by a factor of 868% over the corporate WLAN. Similarly, for 12 clients in the system, the median download throughput for corporate WLAN is 750 Kbps and for DenseAP it is 9.4 Mbps, which is an improvement of over 1250%.

The comparison with the corporate WLAN may seem unfair, because we are comparing the 8-channel, 24-AP DenseAP system against a single-channel, single-AP baseline. However, the only purpose of these results is to show the full benefit of the DenseAP approach in

our testbed. The next step is to separate out the impact of various factors that contribute to these results. As described earlier, the gain in throughput comes from four factors. These are: (i) use of orthogonal channels (ii) dense deployment of APs (iii) use of intelligent associations and (iv) load balancing.

We note that though enabled, the load balancing algorithm played no role in these results. The main reason is that the clients are scattered uniformly across the floor. Thus, in most cases, each client associated with its own DAP. Further, since all clients started at the same time and they all saturated their respective channels, there was no opportunity for our load balancing algorithm to move a client from one DAP to another since all channels were equally loaded. We consider the impact and efficacy of the load balancing algorithm later in Section 6.3.

It is easy to see that more orthogonal channels are better, since the median throughput is higher with 8 channels than with 4 channels. But the important question is whether the DenseAP system derives all its benefit from using more orthogonal channels? That is, can we isolate the impact of the dense deployment of DAPs and our centralized association policy?

To isolate the impact of DAP density, we need to ensure that the number of channels and the association policy play no role in the performance. The way to do this is to evaluate the performance of the DenseAP system with all DAPs operating on the same channel. This experiment is described next.

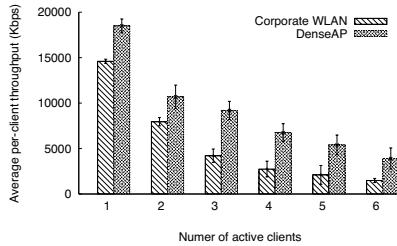


Figure 12. Benefits of density: DenseAP with 1 channel

6.2.3 Using Only 1 channel: Impact of Density

We repeated the previously described experiment with only one channel, and we varied the number of active clients from one to six. We did the experiment for the corporate WLAN, and then repeated it for DenseAP with all DAPs set to use the same channel.

Note that association policy plays very little role in this setting. Our testbed is small, and all DAPs interfere with one another. As a result, load on all DAPs is the same, so the association policy is reduced to simply selecting a DAP that hears the client with reasonable signal strength. For similar reasons, load balancing does not play a role either.

Thus, the only factor providing gains for DenseAP in this setting is the density of the DAPs. The reason density provides performance gains in this setting is the following. As more clients are added, the performance of the corporate WLAN is dominated by the client with the worst connection quality, which is usually the client that is the farthest away from the AP. Due to poor connection quality, such clients use lower transmission rates, thereby consuming more airtime. This, in turn, hurts performance of all other clients. This is known as the rate anomaly problem [13]. With DenseAP however, each client generally talks to a nearby DAP. As a result, clients and DAPs can communicate at higher data rates, thereby reducing the impact of the rate anomaly problem.

The results of this experiment are shown in Figure 12. We see that DenseAP performs better than WLAN even in this setting. The results highlight the benefit of the dense AP deployment. They also explain why, in the previous section, we saw gains of more than 800% with 8 channels!

The results lead us to ask: can a system administrator significantly improve capacity by simply adding more APs to the network? In other words, what is the contribution of the association policy to the overall gain? We examine this in the next section.

6.2.4 Benefits of Association Policy

In this section we demonstrate the benefits of the DenseAP association policy over the client-driven method of association used in conventional WLANs.

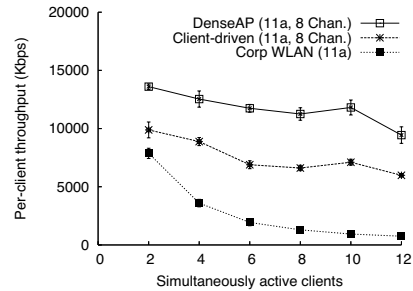


Figure 13. Benefits of the association policy

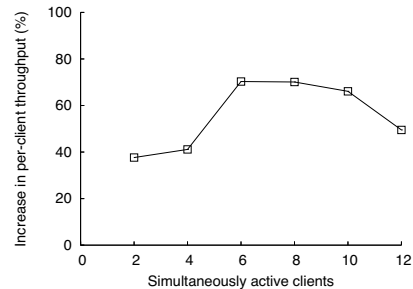


Figure 14. Gains over the client-driven approach

We carried out the following experiment in the 802.11a band, using 8 channels. We disabled the DC. We first assigned channels to all DAPs using the channel assignment algorithm described in [15]. We then disabled the ACLs, and allowed clients to associate with the DAP of their choice. In other words, the association decisions were left to the clients (as it is in today's WLANs). This setup represents a dense deployment with a conventional WLAN approach. We then carried out the experiment described in the previous section.

We note a few points about this particular scenario. There was no pre-existing traffic on any of the channels. Also, the clients were generally evenly distributed across the testbed, and so were the DAPs. Each client then picked the DAP to associate with based on the local client driver implementation policies.

The results of the experiment are compared with the result of running a full fledged DenseAP system with the same deployment and 8 channels. These results, along with the baseline, are shown in Figure 13. The results show that while simply deploying more APs and doing intelligent channel assignment in a conventional WLAN will be beneficial, the benefits will be higher if associations are controlled in a centralized manner.

In other words, the fact that the line labeled "DenseAP" is above the line labeled "Client-driven" is what demonstrates the benefits of the DenseAP approach. The extra gain is due to the intelligent, centralized association control used in the DenseAP system. The magnitude of the extra gain is illustrated in Figure 14. In fact, as we shall see later, the centralized controller can provide roughly the same gains with fewer

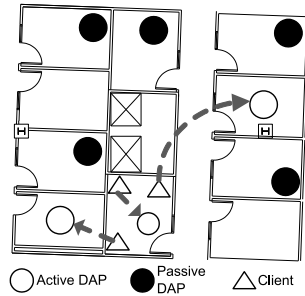


Figure 15. A conference room scenario

APs.

To drive home the point about the benefits of association policy, we consider which DAPs the clients associated with when left to decide by themselves. For example, in the case of 12 active clients, the clients used only 6 channels and 10 APs. On the other hand, by using the association policy, the DenseAP system used all 8 channels, and 11 APs.

One may argue that in the above experiment, the “Client driven” approach performed worse than the DenseAP approach simply because the specific static channel assignment we used for the “Client driven” approach was a bad one. However, we note that *any* static channel assignment algorithm that does not take into account the actual location of clients in the system, is always likely to underperform a dynamic, on-the-fly channel allocation algorithm. We demonstrate this with a simple experiment.

We set up three clients in a small conference room, as shown in Figure 15. There were no other clients in the system. We disabled DC and instead let the three clients pick the DAP to associate with. Unsurprisingly, they all associated with the AP located in the conference room. Note that no static channel assignment algorithm can remedy this situation: the clients *must* associate with different DAPs for channel assignment to have any impact. We repeated the same experiment with the DC enabled and the association policy ensured that the three clients associated with three separate DAPs.

Note however that the association policy alone is not effective. It delivers an improvement in capacity in conjunction with a higher density of DAPs. To demonstrate this, we consider the performance of the system with fewer DAPs.

6.2.5 Performance with Fewer DAPs

We repeated the experiment described in the previous section, but using only 12 of the 24 DAPs deployed in our testbed. The 12 DAPs were selected at random. We used all 8 channels. The results are shown in Figure 16. For comparison purposes, we have also included lines showing the baseline corporate network performance, and the DenseAP performance when using 24 DAPs and 8 chan-

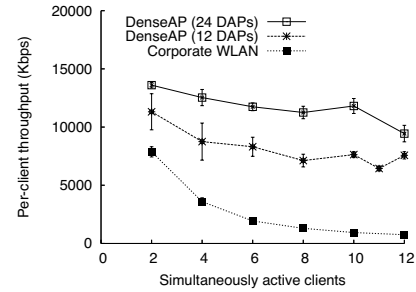


Figure 16. Performance with fewer DAPs

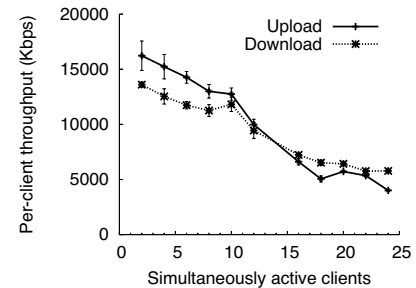


Figure 17. Performance with more clients

nels. As expected, the results show that using fewer DAPs hurts performance.

These results demonstrate the following: (i) More DAPs are beneficial and (ii) the performance of the DenseAP system degrades gracefully, if DAPs were to fail. We have already described how the DC can detect that a DAP has failed, and re-assign its clients to other DAPs. (iii) Note that the performance of the DenseAP system with 12 nodes is similar to the performance of the client-driven approach (Figure 13), with 24 nodes! In other words, the association policy can deliver similar performance with only half as many DAPs.

6.2.6 Performance with More Clients

In all the experiments so far, we have not used more than 12 clients. Since we have deployed 24 DAPs, it is important to consider the performance of the system with more clients. To do this, we extend the experiment described earlier to use up to 24 clients. We use all 24 DAPs, and 8 channels. The upload and download throughputs are shown in Figure 17. We see that the per-client throughput gradually decreases with increasing number of clients. This shows that DenseAP system can gracefully handle the extra load.

6.2.7 Performance with On/Off Traffic

We now turn to more complex traffic patterns as opposed to the throughput of 1-minute TCP flows. Corporate network traffic can be modeled as a series of short

	Mean Time b/w requests	Load Offered per node	Corporate WLAN	DenseAP
Overload	0.5s	2Mbps	828ms	46ms
Full load	1s	1Mbps	187ms	46ms
Low load	2s	0.5Mbps	60ms	46ms

Table 1. Performance with On/Off traffic

flows arriving at various times [12]. The metric of interest for such traffic patterns is the flow completion time.

To compare the performance of DenseAP system with corporate WLAN with such on/off traffic, we carry out the following experiment. We use 12 clients, all of which are active simultaneously. Each client downloads 2000 files from a central server on the corporate network. The sizes of files are chosen from a Pareto distribution with mean of 125KB and shape parameter of 1.5. The time between start of successive downloads is chosen from an exponential distribution with a given mean. By changing the mean time between successive requests, we can control the amount of offered load generated by each client.

We consider three scenarios. In the first scenario, the mean interarrival time between successive downloads is 0.5 seconds. This corresponds to a mean offered load of about 2Mbps per client. In Figure 8, we see that when 12 clients are simultaneously active on the corporate WLAN, the median per-client throughput is 750Kbps. Thus, the 2Mbps offered load represents a heavy overload of the corporate network. We similarly construct a fully loaded, and highly loaded scenario using mean interarrival times of 1 and 2 seconds, respectively. The details are shown in Table 1.

We repeat the experiment on corporate WLAN, as well as the DenseAP system using 24 APs and 8 channels. The median flow completion times under corporate WLAN, and DenseAP system are shown in Table 1. We see that the median flow completion time for corporate WLAN is very high under the overload and full load scenarios. These high flow completion times are detrimental to user experience. On the other hand, in DenseAP the median flow completion time is essentially equal in all three cases, since the load on the system is substantially lower than its capacity.

6.3 Load Balancing

As we have discussed earlier, in our system, a dense deployment of DAPs, coupled with the association policy limits the need for frequent load balancing. Our system uses load balancing only to correct severely imbalanced client-DAP assignments, rather than as a means to achieve “optimal” performance. The reason for this is simple: every client-AP reassignment, no matter how carefully done, carries with it the potential to disrupt a client’s performance.

To illustrate the load balancing capabilities of DenseAP, we carry out the following experiment. We use

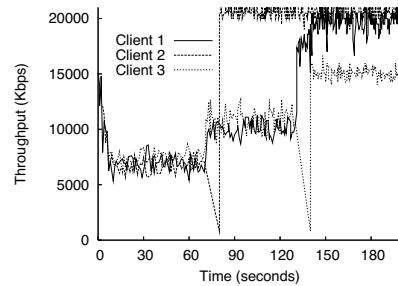


Figure 18. Load balancing: 3 TCP downloads

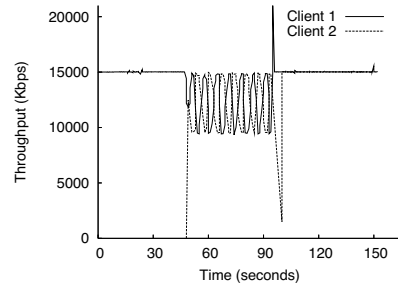


Figure 19. Load balancing: 2 CBR UDP flows

three clients, situated in a conference room, as shown in Figure 15. We force the DC to initially assign all three clients to the conference room DAP. The clients start simultaneously downloading files from a server on the wired network. The DC correctly recognizes the overload situation, and at 1 minute intervals, reassigns two of the associated clients to two nearby APs, and sets them to operate on different channels. As a result, the throughput of all three clients improves substantially. The results are shown in Figure 18.

A similar scenario is shown in Figure 19. We use only two clients. We force the DC to associate both clients to the same AP. The first client starts a CBR UDP download that consumes 15Mbps. This roughly simulates streaming playback of a high quality video. However, this is not enough to saturate the DAP, and hence the DC does not move either client. At time 50 the second client starts a movie download as well. After one minute, the DC detects that an overload situation has occurred and moves the second client to a nearby DAP, and assigns it another channel. The 1-minute hysteresis interval is a tunable parameter of our system, and depending on system configuration and desires of the user population, can be set to a smaller or a larger value.

We now examine the time taken for a handoff from one DAP to another during load balancing. As we will see in the next section, it has significant implications for handling of mobile clients. Recall the sequence of steps for a handoff illustrated in Figure 3. The breakdown of time taken by each of these steps is shown in Table 2. As it has been observed in prior work [23], we see that client scanning is the most expensive step during a hand-

Step	Time (ms)
Disassociation	0
Scanning time	1487.6
Authentication	00.381
Association	00.689
Total handoff time	1488.67

Table 2. Breakdown of a typical handoff in DenseAP.

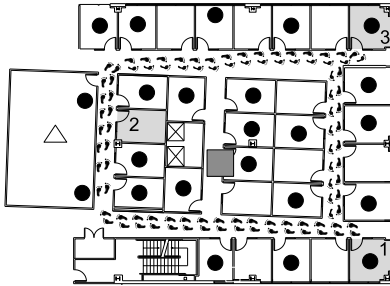


Figure 20. Locations for mobility experiment

off. The 1.5 second delay may cause a TCP timeout, but won't break an existing connection. To mitigate the impact of such disruptions, our load balancing algorithm is very conservative, and moves clients only if they are associated with an overloaded DAP. Such clients would generally be experiencing poor performance in the first place.

6.4 Mobility Experiments

In all the experiments described so far, the clients have been stationary. In this section, we consider how the DenseAP system performs with non-stationary clients. Non-stationary clients fall in two categories, nomadic and mobile.

Nomadic clients move from place to place, but spend significant time being stationary at each place. In corporate WLANs, most non-stationary clients are nomadic clients. A typical example of a nomadic client is an employee who takes her laptop to various meetings. For nomadic clients, the quality of connection they receive when they are "on the move" is less important than the quality of connectivity they receive when they are stationary.

The other type of non-stationary clients are mobile clients. A Wi-Fi VoIP phone user falls in this category. Such clients are rare in current WLANs, but are likely to become more prominent in future [18]. These clients need seamless connectivity as they move. For such clients, metrics such as delay jitter and smoothness of handoff are more important than throughput. Providing good service to mobile clients in a Wi-Fi network is an active topic of research.

Our system can handle both nomadic and mobile clients. We present results for nomadic clients here. The

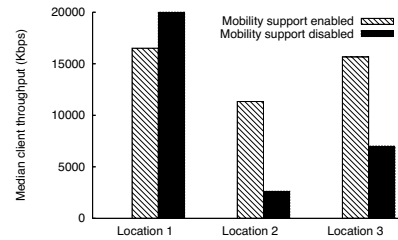


Figure 21. Performance of nomadic client

results for mobile clients are omitted due to lack of space, and are available in [22].

Our system periodically determines the location of each client, and triggers reassociation if the client's position has changed substantially. This works quite well for nomadic clients, since the clients are stationary for most of the time. We demonstrate this with the following experiment.

We setup a client at location 1 on the map shown in Figure 20, and carried out ten 2MB TCP downloads from a server on the wired network. We then walked to location 2, waited for two minutes, and carried out the downloads again. We performed a similar experiment at location 3. We repeated the experiment twice: once without mobility support and once with the support enabled.

The median throughput of the downloads, with and without mobility support is shown in Figure 21. We see that with mobility support enabled, the DC correctly re-associates the client at each location, so its performance does not suffer. Without mobility support, the client continues to be associated with the AP near location 1, and its performance suffers at locations 2 and 3.

6.5 System Scalability

Our architecture uses a central controller (the DC) to manage the DAPs. Each DAPs sends periodic reports to the DC. This raises scalability concerns. To address these concerns, we note that our DC was able to easily manage a network of 24 DAPs and 24 clients, without any special optimizations. The CPU load on the DC never exceeded 30%. We estimate that the amount of control traffic generated by each DAP was less than 20Kbps. Thus, we estimate that a slightly more powerful DC could easily handle a network of about 100 DAPs, without any special optimizations. This should be enough to cover a floor of our office building.

We note here that it is not strictly necessary to use a single central controller. What is necessary is the use of global knowledge while making association and channel assignment decisions. In theory, the functionality of the central controller can be either be replicated, or even implemented in a fully distributed manner. The DAPs can exchange information with each other to gain a global view of the network, and make appropriate decisions.

However, this approach is more complex to implement, and has its own set of scalability concerns.

Another issue we must address is the impact of several DAPs in close vicinity, beaconing and sending probe packets. Our measurements show that in the common case, the impact on performance is less than 1%. This is due to two reasons. First, only active DAPs send beacons, and second, when we use multiple channels, the number of DAPs on any one channel is small.

7 Discussion

We now discuss some issues related to the DenseAP architecture.

Density Re-visited: The density of DAP deployments affects the performance of DenseAP. This raises some important questions that need to be addressed, (i) Where should the DAPs be placed? (ii) Is there a point at which adding more DAPs to the system can hurt performance? (iii) How do we determine the minimum necessary density for a required level of service in a given environment? Guidelines developed for traditional WLANs offer little help in answering these questions, since they are generally developed with an aim of using as few APs as possible while maximizing the coverage area. Question (i) In our current testbed, we distributed the DAPs roughly uniformly in the given area. However, it may be beneficial to deploy more DAPs near “hotspots” such as conference rooms. We are studying this question further. For (ii), thus far, we have demonstrated exploiting density to yield higher gains in capacity. However, with only a finite number of channels and no power control, we expect the benefits from density to diminish beyond a certain point. Mhatre et al. [19] have presented a closed form solution for optimal AP density by varying the CCA threshold, and we are working on validating it on our testbed. To address (iii), we can integrate DAIR [8] with DenseAP to automatically determine RF Holes, i.e. regions with no coverage.

Hidden Terminal: The DenseAP system might exacerbate the hidden terminal problem due to a greater number of parallel transmissions. We have not noticed this effect in our testbed where all DAPs interfere with each other. However, hidden terminals might be a concern in larger testbeds. We are expanding our deployment to investigate this issue in detail. However, our preliminary insight is that the hidden terminal problem might not be severe in the DenseAP scenario because of the capture effect [17]. In a dense deployment of DAPs, the clients are generally located very close to the DAPs they are associated with. Furthermore, the signal in the 5 GHz band fades rapidly in indoor environments thereby reducing the interference from far-away transmitters.

Therefore, we expect the capture effect to reduce the impact of hidden terminal problems.

Spatial Reuse of Channels: When assigning a channel to a DAP, our algorithm can take into account the load on all available channels. The load includes background noise, as well as traffic generated by other DAPs. Thus, we achieve spatial re-use whenever possible. Our algorithm, however, is not optimized to maximize spatial re-use.

Co-existence with Other Wi-Fi Networks: Since we can take the load on a channel into account while assigning channels to DAPs, it is easy to see that DenseAP can co-exist with other Wi-Fi networks. For example, if a nearby network is generating heavy traffic on a particular channel, the DenseAP system can detect it, and avoid assigning that channel to DAPs that are likely to be affected by that network.

What is the Ideal Client-AP Assignment?: The ideal client-AP assignment depends on several issues, including traffic, background noise and environmental factors that affect radio signal propagation. Currently, the DenseAP algorithm ignores the impact of hidden terminal issues, and focuses on avoiding problems such as rate anomaly and AP overloading. We make no claims that our algorithm is optimal. In future, we plan to study the optimality of our algorithm using simulations.

8 Related Work

There has been much prior work on WLAN channel assignment and power control. Several of them [9, 24, 14, 10, 7, 21] either require modifications to the client or to the 802.11 standard. This makes them difficult to deploy. To the best of our knowledge, ours is the first proposal to be built and deployed that performs intelligent associations and deals with a dynamic operating wireless environment *without* requiring client modifications. Of the prior work in this area, we address two systems in particular that come closest to DenseAP.

Similar to DenseAP, MDG [10] identifies intelligent channel assignment, power control and client association as being key components of a systematic approach to increase the capacity of an 802.11 wireless network. It studies the interdependencies between these three knobs and identifies various situations in which a correct order of their application can increase network capacity. Furthermore, MDG modifies clients, and uses explicit feedback and cooperation from them to perform efficient channel assignment, power control and association. In contrast to MDG, DenseAP does not require any modifications to the clients, and therefore explores a different

design space.

SMARTA [7] is similar to DenseAP in that it uses a centralized server to increase the capacity of a dense AP deployment without requiring client modifications. However, it uses a different approach. The central controller builds a conflict graph among the APs, and uses this graph to tune the AP's channel and transmit power. It does not manage client associations. There are two main differences between SMARTA and DenseAP. First, DenseAP relies on correctly managing client associations. We have shown that the benefits of a dense AP deployment is limited if clients are allowed to take association decisions. We have also shown that unilateral power control (without client cooperation) can hurt the performance of the system. We also note that since SMARTA is evaluated entirely in simulations, we are unable to do a fair comparison of SMARTA with our scheme.

In [6], the authors propose using a centralized scheduling mechanism to schedule downlink traffic in a dense deployment of APs. The overall goal is to efficiently manage the data plane of an 802.11 deployment. The work is in progress and at the time of this submission, the authors have not proposed a solution for managing the uplink traffic.

A host of products by networking startup companies [3, 5, 2, 4, 1] are designed to manage AP deployments in the enterprise. The exact details about how their products work are difficult to obtain. However, most systems seem to either ignore association control and load balancing, or they address such challenges by requiring users to install custom drivers.

9 Conclusion

We have demonstrated that DenseAP improves the capacity of an enterprise network. It achieves this by exploiting DAP density via an intelligent association process that encompasses load balancing and dynamic channel allocation. We have described the algorithms and mechanisms necessary to support unmodified clients, and shown significant benefits in a real testbed deployment.

Acknowledgments

We thank Victor Bahl, Ratul Mahajan, Matt Welsh, Aditya Akella, Jon Crowcroft, Lili Qiu, Dina Katabi and the anonymous reviewers for their helpful comments on early drafts of this paper. We thank Mitesh Desai for helping with the SoftAP implementation.

References

- [1] Autocell, <http://www.autocell.com>.

- [2] Bluesocket, <http://www.bluesocket.com>.
[3] Enterprise solutions from aruba networks, <http://www.arubanetworks.com/solutions/enterprise.php>.
[4] Extricom, <http://www.extricom.com>.
[5] Meru networks, <http://www.merunetworks.com>.
[6] N. Ahmed, S. Banerjee, S. Keshav, A. Mishra, K. Papagiannaki, and V. Shrivastava. Interference Mitigation in Wireless LANs using Speculative Scheduling. In *MobiCom*, 2007.
[7] N. Ahmed and S. Keshav. SMARTA: A Self-Managing Architecture for Thin Access Points. In *CoNEXT*, 2006.
[8] P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing the Security of Corporate Wi-Fi Networks Using DAIR. In *MobiSys*, 2006.
[9] P. Bahl, M. Hajiaghayi, K. Jain, V. Mirrokni, L. Qiu, and A. Seber. Cell Breathing in Wireless LANs: Algorithms and Evaluation. *IEEE Transactions on Mobile Computing*, 2006.
[10] I. Broustis, K. Papagiannaki, S. V. Krishnamurthy, M. Faloutsos, and V. Mhatre. MDG: Measurement-driven Guidelines for 802.11 WLAN Design. In *MobiCom*, 2007.
[11] R. Chandra, J. Padhye, A. Wolman, and B. Zill. A Location-based Management System for Enterprise Wireless LANs. In *NSDI*, 2007.
[12] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility Study of Mesh Networks for All-Wireless Offices. In *MobiSys*, 2006.
[13] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance anomaly of 802.11b. In *Infocom*, 2003.
[14] G. Judd and P. Steenkiste. Fixing 802.11 Access Point Selection. In *SIGCOMM Poster Session*, Pittsburgh, PA, July 2002.
[15] B.-J. Ko, V. Misra, J. Padhye, and D. Rubenstein. Distributed channel assignment in multi-radio 802.11 mesh networks. In *WCNC*, 2007.
[16] K. Lakshminarayanan, V. N. Padmanabhan, and J. Padhye. Bandwidth Estimation in Broadband Access Networks. In *IMC*, 2004.
[17] J. Lee, W. Kim, S.-J. Lee, W. Kim, D. Jo, J. Ryu, T. Kwon, and Y. Choi. An Experimental Study on the Capture Effect in 802.11a Networks. In *WINTech*, 2007.
[18] M. Lopez. Forrester Research: The State of North American Enterprise Mobility in 2006. December 2006.
[19] V. Mhatre and K. Papagiannaki. Optimal Design of High Density 802.11 WLANs. In *CoNEXT*, 2006.
[20] V. Mhatre, K. Papagiannaki, and F. Baccelli. Interference Mitigation through Power Control in High Density 802.11 WLANs. In *Infocom*, 2007.
[21] A. Mishra, V. Brik, S. Banerjee, A. Srinivasan, and W. Arbaugh. A Client-driven Approach for Channel Management in Wireless LANs. In *Infocom*, 2006.
[22] R. Murty, R. Chandra, J. Padhye, A. Wolman, and B. Zill. Designing High Performance Enterprise Wi-Fi Networks. Technical Report MSR-TR-2008-28, 2008.
[23] I. Ramani and S. Savage. SyncScan: Practical Fast Handoff for 802.11 Infrastructure Networks. In *Infocom*, Miami, FL, March 2005.
[24] A. Vasan, R. Ramjee, and T. Woo. ECHOS - Enhanced Capacity 802.11 Hotspots. In *Infocom*, 2005.
[25] S. Wong, S. Lu, H. Yang, and V. Bharghavan. Robust rate adaptation for 802.11 wireless networks. In *MobiCom*, 2006.

FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput

SRIKANTH KANDULA
MIT

KANDULA@MIT.EDU

KATE CHING-JU LIN
NTU/MIT

KATE@CSAIL.MIT.EDU

TURAL BADIRKHANLI
MIT

TURALB@MIT.EDU

DINA KATABI
MIT

DK@MIT.EDU

Abstract— It is increasingly common that computers in residential and hotspot scenarios see multiple access points (APs). These APs often provide high speed wireless connectivity but access the Internet via independent, relatively low-speed DSL or cable modem links. Ideally, a client would simultaneously use all accessible APs and obtain the sum of their backhaul bandwidth. Past work can connect to multiple APs, but can neither aggregate AP backhaul bandwidth nor can it maintain concurrent TCPs across them.

This paper introduces FatVAP, an 802.11 driver that aggregates the bandwidth available at accessible APs and also balances their loads. FatVAP has three key features. First, it chooses the APs that are worth connecting to and connects with each AP just long enough to collect its available bandwidth. Second, it ensures fast switching between APs without losing queued packets, and hence is the only driver that can sustain concurrent high throughput TCP connections across multiple APs. Third, it works with unmodified APs and is transparent to applications and the rest of the network stack. We experiment with FatVAP both in our lab and hotspots and residential deployments. Our results show that, in today's deployments, FatVAP immediately delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.

1 INTRODUCTION

Today, WiFi users often see many access points (APs), multiple of which are open [10], or accessible at a small charge [9]. The current 802.11 connectivity model, which limits a user to a single AP, cannot exploit this phenomenon and, as a result, misses two opportunities.

- It does not allow a client to harness unused bandwidth at multiple APs to maximize its throughput. Users in hotspots and residential scenarios typically suffer low throughput, despite the abundance of high-speed APs. This is because these high-speed APs access the Internet via low capacity (e.g., 1Mb/s or less) DSL or cable modem links. Since the current connectivity model ties a user to a single AP, a user's throughput at home or in a hotspot is limited by the capacity of a single DSL line, even when there are plenty of high-speed

APs with underutilized DSL links.

- It does not facilitate load balancing across APs. WiFi users tend to gather in a few locations (e.g., a conference room, or next to the window in a cafe). The current 802.11 connectivity model maps all of these users to a single AP, making them compete for the same limited resource, even when a nearby AP hardly has any users [12, 24]. Furthermore, the mapping is relatively static and does not change with AP load.

Ideally, one would like a connectivity model that approximates a *fat virtual AP*, whose backhaul capacity is the sum of the access capacities of nearby APs. Users then compete fairly for this fat AP, limited only by security restrictions. A fat AP design benefits users because it enables them to harness unused bandwidth at accessible APs to maximize their throughput. It also benefits AP owners because load from users in a campus, office, or hotel is balanced across all nearby APs, reducing the need to install more APs.

It might seem that the right strategy to obtain a fat virtual AP would be to greedily connect to every AP. However, using all APs may not be appropriate because of the overhead of switching between APs. In fact, if we have to ensure that TCP connections simultaneously active across multiple APs do not suffer timeouts, it might be impossible to switch among all the APs. Also, all APs are not equal. Some may have low load, others may have better backhaul capacities or higher wireless rates (802.11a/g vs. 802.11b). So, a client has to ascertain how valuable an AP is and spend more time at APs that it is likely to get more bandwidth from, i.e., the client has to divide its time among APs so as to maximize its throughput. Further, the efficiency of any system that switches between APs on short time scales crucially depends on keeping the switching overhead as low as possible. We need a system architecture that not only shifts quickly between APs, but also ensures that no in-flight packets are lost in the process.

While prior work virtualizes a wireless card allowing it to connect to multiple APs, card virtualization alone cannot approximate a fat virtual AP. Past work uses this virtualization to bridge a WLAN with an ad-hoc network [6, 13], or debug wireless connectivity [11], but

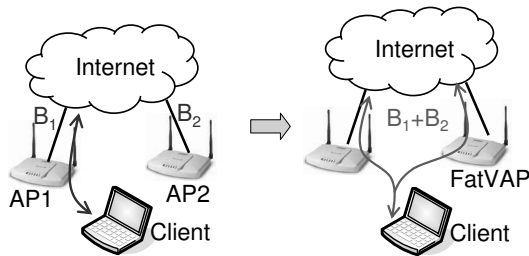


Figure 1: An example scenario where a client can potentially obtain the sum of the backhaul bandwidth available at the two APs.

can neither aggregate AP backhaul bandwidth nor balance their load. This is because it cannot tell which APs are worth connecting to and for how long. Further, it has a large switching overhead of 30-600ms [7, 13] and hence cannot be used for switching at short time-scales on the order of 100 ms, which is required for high-throughput TCP connections across these APs.

This paper introduces FatVAP, an 802.11 driver design that enables a client to aggregate the bandwidth available at accessible APs and balance load across them. FatVAP approximates the concept of a fat virtual AP given the physical restrictions on the resources. To do so, FatVAP periodically measures both the wireless and end-to-end bandwidths available at each AP. It uses this information as well as an estimate of the switching overhead to connect to each AP for just enough time to collect its available bandwidth and toggle only those APs that maximize user throughput.

The FatVAP driver has the following key features.

- It has an AP scheduler that chooses how to distribute the client's time across APs so as to maximize throughput.
- It ensures fast switching between APs (about 3 ms) without losing queued packets, and hence is the only driver that can sustain concurrent high throughput TCP connections across multiple APs.
- It works with existing setups, i.e., single 802.11 card, unmodified APs, and is transparent to applications and the rest of the network stack.

FatVAP leverages today's deployment scenarios to provide immediate improvements to end users without any modification to infrastructure or protocols. It does not need fancy radios, access to the firmware, or changes to the 802.11 MAC. FatVAP has been implemented in the MadWifi driver [4], and works in conjunction with autorate algorithms, carrier-sense, CTS-to-self protection, and all other features in the publicly released driver.

Experimental evaluation of our FatVAP prototype in a testbed and actual hotspot deployments shows that:

- In today's residential and Hotspot deployments (in Cambridge/Somerville MA), FatVAP immediately delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.

- FatVAP is effective at harnessing unused bandwidth from nearby APs. For example, with 3 APs bottlenecked at their backhaul links, FatVAP's throughput is 3x larger than an unmodified MadWifi driver.
- FatVAP effectively balances AP loads. Further, it adapts to changes in the available bandwidth at an AP and re-balances load with no perceivable delay.
- FatVAP coexists peacefully. At each AP, FatVAP competes with unmodified clients as fairly as an unmodified MadWifi driver and is sometimes fairer as FatVAP will move to an alternate if the AP gets congested. Further, FatVAP clients compete fairly among themselves.

2 MOTIVATING EXAMPLES

Not all access points are equal. An 802.11 client might have a low loss-rate to one access point; another access point might be less congested; yet another may have a high capacity link to the Internet or support higher data rates (802.11g rather than 802.11b). How should an 802.11 client choose which access points to connect to and what fraction of its time to stay connected to each AP?

To better understand the tradeoffs in switching APs, let us look at a few simple examples. Consider the scenario in Fig. 1, where the wireless client is in the range of 2 open APs. Assume the APs operate on orthogonal 802.11 channels. For each AP, let the *wireless available bandwidth*, w , be the rate at which the client communicates with the AP over the wireless link, and the *end-to-end available bandwidth*, e , be the client's end-to-end data rate when connected to that AP. Note that these values do not refer to link capacities but the throughput achieved by the client and in particular subsume link losses, driver's rate selection and competition from other clients at the AP. Note also that the end-to-end bandwidth is always bounded by the wireless available bandwidth, i.e., $e \leq w$. How should the client in Fig. 1 divide its time between connecting to AP1 and AP2? The answer to this question depends on a few factors.

First, consider a scenario in which the bottlenecks to both APs are the wireless links (i.e., $w = e$ at both APs). In this case, there is no point toggling between APs. If the client spends any time at the AP with lower available wireless bandwidth, it will have to send at a lower rate for that period, which reduces the client's overall throughput. Hence, when the wireless link is the bottleneck, the client should stick to the best AP and avoid AP switching.

Now assume that the bottlenecks are the APs' access links (i.e., $w > e$ for both APs). As a concrete example, say that the client can achieve 5 Mb/s over either wireless link, i.e., $w_1 = w_2 = 5$ Mb/s, but the client's end-to-end available bandwidth across either AP is only 2 Mb/s, i.e., $e_1 = e_2 = 2$ Mb/s. If the client picks one of the two

AP Bandwidth (Mbps)	AP1	AP2	AP3
End-to-end Available	5	4	3
Wireless Available	5	8	8

Optimal = 7 Mbps, 88% busy

Figure 2: Choosing APs greedily based on higher end-to-end available bandwidth is not optimal.

AP Bandwidth (Mbps)	AP1	AP2	AP3	AP4	AP5	AP6
End-to-end Available	1	1	1	1	1	4.5
Wireless Available	5	5	5	5	5	4.5

Figure 3: Choosing APs greedily based on higher wireless available bandwidth is not practical because it doesn't account for switching overheads.

APs and sticks to it, as is the case with current drivers, its throughput will be limited to 2 Mb/s. We observe however that the client need not spend 100% of its time at an AP to obtain its end-to-end available bandwidth. It is sufficient to connect to each AP for $\frac{2}{5}$ of the client's time. While connected, the client sends (and receives) its data at 5 Mb/s, i.e., according to its wireless available bandwidth. The AP drains the client's data upstream (or receives new data for the client) at the lower rate of 2 Mb/s, which is the end-to-end bandwidth available to our client. Until the AP drains the previous burst (or gets new data for the client), there is no point for the client to stay connected to the AP. As long as the client spends more than $\frac{2}{5}$ of its time on each AP, it can achieve the sum of their end-to-end rates, i.e., in our example it achieves 4 Mb/s.

Thus, to obtain the bandwidth available at an AP, a client should connect to it for at least a fraction $f_i = \frac{e_i}{w}$ of its time. This means that when the wireless link is the bottleneck at an AP, i.e., $w = e$, a client needs to spend 100% of its time at that AP in order to collect its available bandwidth. Otherwise, the client can use its spare time to get then unused bandwidth at other APs. But since the sum of the f_i 's across all APs can exceed 1, a client will need to select a subset of the available APs. So, which APs does a client pick?

One may think of making greedy decisions. In particular, the client can order the APs according to their end-to-end available bandwidth, and greedily add APs to its schedule until the sum of the fractions f_i 's reaches 1—i.e., 100% of the client's time is used up. Such a scheduler however is suboptimal. Fig. 2 shows a counter example, where AP1 has the highest end-to-end rate of 5Mb/s, yet picking AP1 means that the client has to spend $\frac{e_i}{w_i} = \frac{5}{5} = 100\%$ of its time at AP1 leaving no time to connect to other APs. The optimal scheduler here picks {AP2, AP3} and achieves 7 Mb/s throughput; the client spends $\frac{e_i}{w_i} = \frac{4}{8} = 50\%$ of its time at AP2 and $\frac{3}{8} = 38\%$ at AP3 for a total of 88% of busy time. The remaining 12% of time can compensate for the switching overhead and increase robustness to inaccurate estimates of AP bandwidth.

In practice, one also cannot pick APs greedily based on their wireless available bandwidth. Consider the example in Fig. 3. One may think that the client should toggle between AP1, AP2, AP3, AP4, and AP5, spending 20% of its time on each AP. This would have been true if switching APs takes no time. In practice, switching between APs incurs a delay to reset the hardware to a different channel, to flush packets within the driver, etc., and this overhead adds up over the number of APs switched. Consider again the scenario in Fig. 3. Let the switching delay be 5 ms, then each time it toggles between 5 APs, the client wastes 25 ms of overhead. This switching overhead cannot be amortized away by switching infrequently between APs. To ensure that TCP connections via an AP do not time out, the client needs to serve each AP frequently, say once every 100ms. With a duty cycle of 100ms, and a switching overhead of 25ms a client has only 75% of its time left for useful work. Dividing this over the five APs results in a throughput of $5 \times .75 = 3.75$ Mb/s, which is worse than sticking to AP6 for 100% of the time, and obtaining 4.5 Mb/s.

In §3.1, we formalize and solve a scheduling problem that maximizes client throughput given practical constraints on switching overhead and the switching duty cycle.

3 FATVAP

FatVAP is an 802.11 driver design that aggregates the bandwidth available at nearby APs and load balances traffic across them. We implemented FatVAP as a modification to the MadWifi driver [4]. FatVAP incorporates the following three components:

- An AP scheduler that maximizes client throughput;
- A load balancer that maps traffic to APs according to their available bandwidth;
- An AP switching mechanism that is fast, loss-free, and transparent to both the APs and the host network stack.

At a high level, FatVAP works as follows. FatVAP scans the various channels searching for available access-points (APs). It probes these APs to estimate their wireless and end-to-end available bandwidths. FatVAP's scheduler decides which APs are worth connecting to and for how long in order to maximize client throughput. FatVAP then toggles connections to APs in accordance to the decision made by the scheduler. When switching away from an AP, FatVAP informs the AP that the client is entering the power-save mode. This ensures that the AP buffers the client's incoming packets, while it is away collecting traffic from another AP. Transparent to user's applications, FatVAP pins flows to APs in a way that balances their loads. FatVAP continually estimates the end-to-end and wireless available bandwidths at each

AP by passively monitoring ongoing traffic, and adapts to changes in available bandwidth by re-computing the best switching schedule.

3.1 The AP Scheduler

The scheduler chooses which APs to toggle between to maximize client throughput, while taking into account the bandwidth available at the APs and the switching overhead.

We formalize the scheduling problem as follows. The scheduler is given a set of accessible APs. It assigns to each AP a value and a cost. The value of connecting to a particular AP is its contribution to client throughput. If f_i is the fraction of time spent at AP_{*i*}, and w_i is AP_{*i*}'s wireless available bandwidth, then the value of connecting to AP_{*i*} is:

$$\text{value}_i = f_i \times w_i. \quad (1)$$

Note that as discussed in §2, a client can obtain no more than the end-to-end available bandwidth at AP_{*i*}, e_i , and thus need not connect to AP_{*i*} for more than $\frac{e_i}{w_i}$ of its time. Hence,

$$0 \leq f_i \leq \frac{e_i}{w_i} \Rightarrow \text{value}_i \leq e_i. \quad (2)$$

The cost of an AP is equal to the time that a client has to spend on it to collect its value. The cost also involves a setup delay to pick up in-flight packets and retune the card to a new channel. Note that the setup delay is incurred only when the scheduler spends a non-zero amount of time at AP_{*i*}. Hence, the cost of AP_{*i*} is:

$$\text{cost}_i = f_i \times D + \lceil f_i \rceil \times s, \quad (3)$$

where D is the scheduler's duty cycle, i.e., the total time to toggle between all scheduled APs, s is the switching setup delay, and $\lceil f_i \rceil$ is the ceiling function, which is one if $f_i > 0$ and zero otherwise.

The objective of the scheduler is to maximize client throughput. The scheduler, however, cannot have too large a duty cycle. If it did, the delay can hamper the TCP connections, increasing their RTTs, causing poor throughput and potential time-outs. The objective of the scheduler is to pick the f_i 's to maximize the switching value subject to two constraints: the cost in time must be no more than the chosen duty cycle, D , and the fraction of time at an AP has to be positive and no more than $\frac{e_i}{w_i}$, i.e.,

$$\max_{f_i} \quad \sum_i f_i w_i \quad (4)$$

$$\text{s.t.} \quad \sum_i (f_i D + \lceil f_i \rceil s) \leq D \quad (5)$$

$$0 \leq f_i \leq \frac{e_i}{w_i}, \forall i. \quad (6)$$

How do we solve this optimization? In fact, the optimization problem in Eqs. 4-6 is similar to the known

knapsack problem [3]. Given a set of items, each with a value and a weight, we would like to pack a knapsack so as to maximize the total value subject to a constraint on the total weight. Our items (the APs) have both fractional weights (costs) $f_i \times D$ and zero-one weights $\lceil f_i \rceil \times s$. The knapsack problem is typically solved using dynamic programming. The formulation of this dynamic programming solution is well-known and can be used for our problem [3].

A few points are worth noting.

- FatVAP's solution based on dynamic programming is efficient and stays within practical bounds. Even with 5 APs, our implementation on a 2GHz x86 machine solves the optimization in 21 microseconds (see §4.2).
- So far we have assumed that we know both the wireless and end-to-end bandwidths of all accessible APs. FatVAP estimates these values passively (§3.1.1, §3.1.2).
- The scheduler takes AP load into account. Both the wireless and end-to-end bandwidths refer to the rate obtained by the client as it competes with other clients.
- It is important to include the switching overhead, s , in the optimization. This variable accounts for various overheads such as switching the hardware, changing the driver's state, and waiting for in-flight packets. It also ensures that the scheduler shies away from switching APs whenever a tie exists, or when switching does not yield a throughput increase. FatVAP continuously measures the switching delay and updates s if the delay changes (we show microbenchmarks in §4.2).
- Our default choice for duty cycle is $D = 100$ ms. This value is long enough to enable the scheduler to toggle a handful of APs and small enough to ensure that the RTTs of the TCP flows stay in a reasonable range [19].

3.1.1 Measuring Wireless Available Bandwidth

The wireless available bandwidth is the rate at which the client and AP communicate over the wireless link. If the client is the only contender for the medium, the wireless available bandwidth is the throughput of the wireless link. If other clients are contending for the medium, it reduces to the client's competitive share of the wireless throughput after factoring in the effect of auto-rate. Here, we describe how to estimate the wireless available bandwidth from client to the AP, i.e., on the uplink. One can have separate estimates for uplink and downlink. However, in our experience the throughput gain from this improved accuracy is small in comparison to the extra complexity.

How does a client estimate the uplink wireless available bandwidth? The client can estimate it by measur-

ing the time between when a packet reaches the head of the transmit queue and when the packet is acked by the AP. This is the time taken to deliver one packet, td , given contention for the medium, autorate, retransmissions, etc. We estimate the available wireless bandwidth by dividing the packet's size in bytes, B , by its delivery time td . The client takes an exponentially weighted average over these measurements to smooth out variability, while adapting to changes in load and link quality.

Next, we explain how we measure the delivery time td . Note that the delivery time of packet j is:

$$td_j = ta_j - tq_j, \quad (7)$$

where tq_j is the time when packet j reaches the head of the transmit queue, and ta_j is the time when packet j is acked. It is easy to get ta_j because the Hardware Abstraction Layer (HAL) timestamps each transmitted packet with the time it was acked. Note that the HAL does raise a tx interrupt to tell the driver to clean up the resources of transmitted packets but it does this only after many packets have been transmitted. Hence, the time when the tx interrupt is raised is a poor estimate of ta_j .

Obtaining tq_j , however, is more complex. The driver hands the packet to the HAL, which queues it for transmission. The driver does not know when the packet reaches the head of the transmission queue. Further, we do not have access to the HAL source, so we cannot modify it to export the necessary information.¹ We work around this issue as follows. We make the driver timestamp packets just before it hands them to the HAL. Suppose the timestamp of packet j as it is pushed to the HAL is th_j , we can then estimate tq_j as follows:

$$tq_j = \max(th_j, ta_{j-1}) \quad (8)$$

The intuition underlying Eq. 8 is simple; either the HAL's queue is empty and thus packet j reaches the head of the queue soon after it is handed to the HAL, i.e., at time th_j , or the queue has some previous packets, in which case packet j reaches the head of the queue only when the HAL is done with delivering packet $j - 1$, i.e., at time ta_{j-1} .

Two practical complications exist however. First, the timer in the HAL has millisecond accuracy. As a result, the estimate of the delivery time td in Eq. 7 will be equally coarse, and mostly either 0 ms or 1 ms. To deal with this coarse resolution, we need to aggregate over a large number of measurements. In particular, FatVAP produces a measurement of the wireless available throughput at AP_i by taking an average over a window of T seconds (by default $T = 2$ s), as follows:

$$w_i = \frac{\sum_{j \in T} B_j}{\sum_{j \in T} td_j}. \quad (9)$$

¹An open source project named OpenHAL allows access to the HAL but is too inefficient to be used in practice.

The scheduler continuously updates its estimate by using an exponentially weighted average over the samples in Eq. 9.

A second practical complication occurs because both the driver's timer and the HAL's timer are typically synchronized with the time at the AP. This synchronization happens with every beacon received from the AP. But as FatVAP switches APs, the timers may resynchronize with a different AP. This is fine in general as both timers are always synchronized with respect to the same AP. The problem, however, is that some of the packets in the transmit queue may have old timestamps taken with respect to the previous AP. To deal with this issue, the FatVAP driver remembers the id of the last packet that was pushed into the HAL. When resynchronization occurs (i.e., the beacon is received), it knows that packets with ids smaller than or equal to the last pushed packet have inaccurate timestamps and should not contribute to the average in Eq. 9.

Finally, we note that FatVAP's estimation of available bandwidth is mostly passive and leverages transmitted data packets. FatVAP uses probes only during initialization, because at that point the client has no traffic traversing the AP. FatVAP also occasionally probes the unused APs (i.e., APs not picked by the scheduler) to check that their available bandwidth has not changed.

3.1.2 Measuring End-to-End Available Bandwidth

The available end-to-end bandwidth via an AP is the average throughput that a client obtains when using the AP to access the Internet.² The available end-to-end bandwidth is lower when there are more contenders causing a FatVAP client to avoid congested APs in favor of a balanced load.

How do we measure an AP's end-to-end available bandwidth? The naive approach would count all bytes received from the AP in a certain time window and divide the count by the window size. The problem, however, is that no packets might be received either because the host has not demanded any, or the sending server is idle. To avoid underestimating the available bandwidth, FatVAP guesses which of the inter-packet gaps are caused by idleness and removes those gaps. The algorithm is fairly simple. It ignores packet gaps larger than one second. It also ignores gaps between small packets, which are mostly AP beacons and TCP acks, and focuses on the spacing between pairs of large packets. After ignoring packet pairs that include small packets and those that are spaced by excessively long intervals, FatVAP computes an estimate

²Note that our definition of available end-to-end bandwidth is not the typical value [17, 26] that is computed between a source-destination pair, but is an average over all paths through the AP.

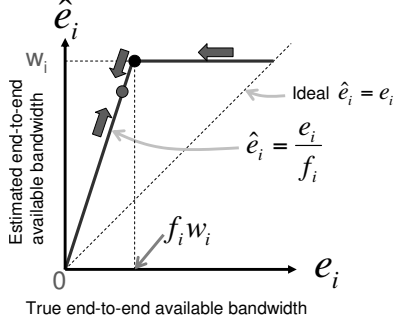


Figure 4: The estimate of end-to-end available bandwidth is different from the true value because the AP buffers data when the client is not listening and buffered data drains at the wireless available bandwidth. FatVAP corrects for this by spending slightly longer than necessary at each AP, i.e., operating at the red dot rather than the black dot.

of the end-to-end available bandwidth at AP_i as:

$$\hat{e}_i = \frac{\sum B_j}{\sum g_j}, \quad (10)$$

where B_j is the size of the second packet in the j^{th} pair, and g_j is the gap separating the two packets, and the sum is taken over a time window of $T = 2$ seconds.

One subtlety remains however. When a client reconnects to an AP, the AP first drains out all packets that it buffered when the client was away. These packets go out at the wireless available bandwidth w_i . Once the buffer is drained out, the remaining data arrives at the end-to-end available bandwidth e_i . Since the client receives a portion of its data at the wireless available bandwidth and $w_i \geq e_i$, simply counting how quickly the bytes are received, as in Eq. 10, over-estimates the end-to-end available bandwidth.

Fig. 4 plots how the estimate of end-to-end available bandwidth \hat{e}_i relates to the true value e_i . There are two distinct phases. In one phase, the estimate is equal to w_i , which is shown by the flat part of the solid blue line. This phase corresponds to connecting to AP_i for less time than needed to collect all buffered data, i.e., $f_i < \frac{e_i}{w_i}$. Since the buffered data drains at w_i , the estimate will be $\hat{e}_i = w_i$. In the other phase, the estimate is systematically inflated by $\frac{1}{f_i}$, as shown by the tilted part of the solid blue line. This phase corresponds to connecting to AP_i for more time than needed to collect all buffered data, i.e., $f_i > \frac{e_i}{w_i}$. The derivation for this inflation is in Appendix A. Here, we note the ramifications.

Inflated estimates of the end-to-end available bandwidth make the ideal operating point unstable. A client would ideally operate at the black dot in Fig. 4, where it connects to AP_i for exactly $f_i^* = \frac{e_i}{w_i}$ of its time. But, if the client does so, the estimate \hat{e}_i will be $\hat{e}_i = \frac{e_i}{f_i^*} = w_i$. In this case, the client cannot figure out the amount of inflation in e_i and compensate for it because the true end-to-end available bandwidth can be any value corresponding to the flat thick blue line in Fig. 4. Even worse, if the actual end-to-end available bandwidth were to increase, say

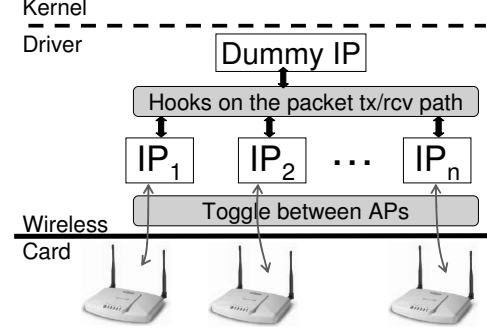


Figure 5: FatVAP's reverse NAT architecture.

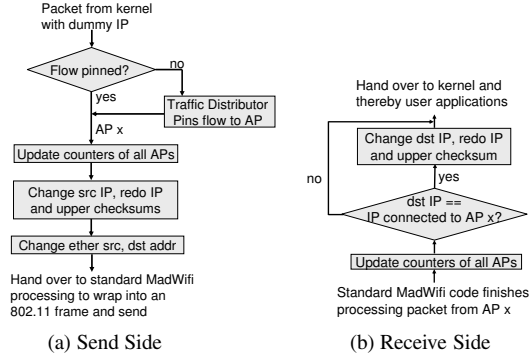


Figure 6: Getting packets to flow over multiple interfaces.

because a contending client shuts off, the client cannot observe this change, because its estimate will still be w_i .

To fix this, FatVAP clients operate at the red dot, i.e., they spend slightly longer than necessary at each AP in order to obtain an accurate estimate of the end-to-end bandwidth. Specifically, if $\hat{e}_i \approx w_i$, FatVAP knows that it is operating near or beyond the black dot and thus slightly increases f_i to go back to the red dot. The red arrows in the figure show how a FatVAP client gradually adapts its f_i to bring it closer to the desired range. As long as f_i is larger than the optimal value, we can compensate for the inflation knowing that $e_i = f_i \hat{e}_i$, i.e., Eq. 10 can be re-written as:

$$e_i = f_i \frac{\sum B_j}{\sum g_j}. \quad (11)$$

3.2 Load Balancing Traffic Across APs

The scheduler in §3.1 gives an opportunity to obtain the sum of available bandwidth at all APs, but to fulfill that opportunity, the FatVAP driver should map traffic to APs appropriately. There are two parts to mapping traffic: a load balancer that splits traffic among the APs, and a reverse-NAT that ensures traffic goes through the desired APs.

3.2.1 The Load Balancer

The load balancer assigns traffic to APs proportionally to the end-to-end bandwidth obtainable from an AP.

Thus, the traffic ratio assigned to each AP, r_i , is:

$$r_i = \frac{f_i w_i}{\sum_j f_j w_j}, \quad (12)$$

where f_i is the fraction of time that the client connects to AP_{*i*} and $f_i w_i$ is the value of AP_{*i*} (see Eqs. 1, 2).

When splitting traffic, the first question is whether the traffic allocation unit should be a packet, a flow, or a destination? FatVAP allocates traffic to APs on a flow-by-flow basis. A flow is identified by its destination IP address and its ports. FatVAP records the flow-to-AP mapping in a hash-table. When a new flow arrives, FatVAP decides which AP to assign this flow to and records the assignment in the hash table. Subsequent packets in the flow are simply sent through the AP recorded in the hash table.

Our decision to pin flows to APs is driven by practical considerations. First, it is both cumbersome and inefficient to divide traffic at a granularity smaller than a flow. Different APs usually use different DHCP servers and accept traffic only when the client uses the IP address provided by the AP's DHCP server. This means that in the general case, a flow cannot be split across APs. Further, splitting a TCP flow across multiple paths often reorders the flow's packets hurting TCP performance [25]. Second, a host often has many concurrent flows, making it easy to load balance traffic while pinning flows to APs. Even a single application can generate many flows. For example, browsers open parallel connections to quickly fetch the objects in a web page (e.g., images, scripts) [18], and file-sharing applications like BitTorrent open concurrent connections to peers.

But, how do we assign flows to APs to satisfy the ratios in Eq. 12? The direct approach assigns a new flow to the i^{th} AP with a random probability r_i . Random assignment works when the flows have similar sizes. But flows vary significantly in their sizes and rates [15, 22, 25]. To deal with this issue, FatVAP maintains per-AP token counters, C , that reflect the deficit of each AP, i.e., how far the number of bytes mapped to an AP is from its desired allocation. For every packet, FatVAP increments all counters proportionally to the APs' ratios in Eq. 12. The counter of the AP that the packet was sent/received on is decremented by the packet size B . Hence, every window of T_c seconds (default is $T = 60$ s) we compute:

$$C_i = \begin{cases} C_i + r_i \times B - B & \text{Packet is mapped to AP}_i \\ C_i + r_i \times B & \text{Otherwise.} \end{cases} \quad (13)$$

It is easy to see that APs with more traffic than their fair share have negative counters and those with less than their fair share have positive counter values. When a new flow arrives, FatVAP assigns the flow to the AP with the most positive counters and decreases that AP's counters

by a constant amount F (default 10,000) to accommodate for TCP's slow ramp-up. Additionally, we decay all counters every $T_c = 60$ s to forget biases that occurred a long time ago.

3.2.2 The Reverse-NAT

How do we ensure that packets in a particular flow are sent and received through the AP that the load balancer assigns the flow to? If we simply present the kernel with multiple interfaces, one interface per AP like prior work [13], the kernel would send all flows through one AP. This is because the kernel maps flows to interfaces according to routing information, not load. When all APs have valid routes, the kernel simply picks the default interface.

To address this issue, FatVAP uses a reverse NAT as a shim between the APs and the kernel, as shown in Fig. 5. Given a single physical wireless card, the FatVAP driver exposes just one interface with a dummy IP address to the kernel. To the rest of the MadWifi driver, however, FatVAP pretends that the single card is multiple interfaces. Each of the interfaces is associated to a different AP, using a different IP address. Transparent to the host kernel, FatVAP resets the addresses in a packet so that the packet can go through its assigned AP.

On the send side, and as shown in Fig. 6, FatVAP modifies packets just as they enter the driver from the kernel. If the flow is not already pinned to an AP, FatVAP uses the load balancing algorithm above to pin this new flow to an AP. FatVAP then replaces the source IP address in the packet with the IP address of the interface that is associated with the AP. Of course, this means that the IP checksum has to be re-done. Rather than recompute the checksum of the entire packet, FatVAP uses the fact that the checksum is a linear code over the bytes in the packet. So analogous to [14], the checksum is recomputed by subtracting some f (the dummy IP address) and adding f (assigned interface's IP). Similarly, transport layer checksums, e.g., TCP and UDP checksums, need to be redone as these protocols use the IP header in their checksum computation. After this, FatVAP hands over the packet to standard MadWifi processing, as if this were a packet the kernel wants to transmit out of the assigned interface.

On the receive side, FatVAP modifies packets after standard MadWifi processing, just before they are handed up to the kernel. If the packet is not a broadcast packet, FatVAP replaces the IP address of the actual interface the packet was received on with the dummy IP of the interface the kernel is expecting the packets on. Checksums are re-done as on the send side, and the packet is handed off to the kernel.

3.3 Fast, Loss-Free, and Transparent AP Switching

To maximize user throughput, FatVAP has to toggle between APs according to the scheduler in §3.1 while simultaneously maintaining TCP flows through multiple APs (see §3.2). Switching APs requires switching the HAL and potentially resetting the wireless channel. It also requires managing queued packets and updating the driver's state. These tasks take time. For example, the Microsoft virtual WiFi project virtualizes an 802.11 card, allowing it to switch from one AP to another. But this switching takes 30-600 ms [7] mostly because a new driver module needs to be initialized when switching to a new AP. Though successful in its objective of bridging wireless networks, the design of Virtual WiFi is not sufficient to aggregate AP bandwidth. FatVAP needs to support fast AP switching, i.e., a few milliseconds, otherwise the switching overhead may preclude most of the benefits. Further, switching should not cause packet loss. If the card or the AP loses packets in the process, switching will hurt TCP traffic [25]. Finally, most of the switching problems would be easily solved if one can modify both APs and clients. Such a design, however, will not be useful in today's 802.11 deployments.

3.3.1 Fast and Loss-Free Switching

The basic technique that enables a card to toggle between APs is simple and is currently used by the MadWiFi [4] driver to background scan for better APs and others. Before a client switches away from an AP, it tells the AP that it is going to power save mode. This causes the AP to buffer the client's packets for the duration of its absence. When the client switches again to the AP, it sends the AP a frame to inform the AP of its return, and the AP then, forwards the buffered packets.

So, how do we leverage this idea for quickly switching APs without losing packets? Two fundamental issues need to be solved. First, when switching APs, what does one do with packets inside the driver destined for the old AP? An AP switching system that sits outside the driver, like MultiNet [13] has no choice but to wait until all packets queued in the driver are drained, which could take a while. Systems that switch infrequently, such as MadWifi that does so to scan in the background, drop all the queued packets. To make AP switching fast and loss-free, FatVAP pushes the switching procedure to the driver, where it maintains multiple transmit queues, one for each interface. Switching APs simply means detaching the old AP's queue and reattaching the new AP's queue. This makes switching a roughly constant time operation and avoids dropping packets. It should be noted that packets are pushed to the transmit queue by the driver and read by the HAL. Thus, FatVAP still needs to wait to resolve the state of the head of the queue. This is, however, a much

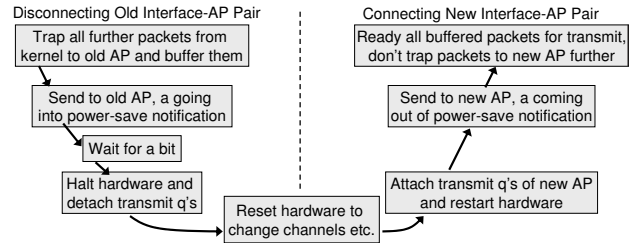


Figure 7: FatVAP's approach to switching between interfaces

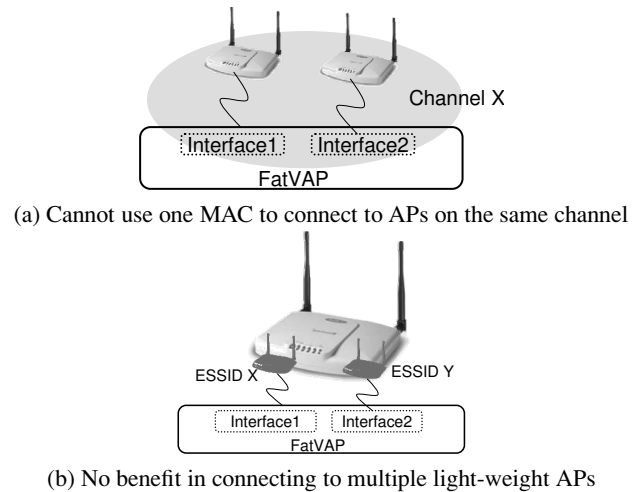


Figure 8: Challenges in transparently connecting to multiple APs.

shorter wait (a few milliseconds) with negligible impact on TCP and the scheduler.

Second, how do we maintain multiple 802.11 state machines simultaneously within a single driver? Connecting with an AP means maintaining an 802.11 state machine. For example, in 802.11, a client transitions from INIT to SCAN to AUTH to ASSOC before reaching RUN, where it can forward data through the AP. It is crucial to handle state transitions correctly because otherwise no communication may be possible. For example, if an association request from one interface to its AP is sent out when another interface is connected to its AP, perhaps on a different channel, the association will fail preventing further communication. To maintain multiple state machines simultaneously, FatVAP adds hooks to MadWifi's 802.11 state-machine implementation. These hooks trap all state transitions in the driver. Only transitions for the interface that is currently connected to its AP can proceed, all other transitions are held pending and handled when the interface is scheduled next. Passive changes to the state of an interface such as receiving packets or updating statistics are allowed at all times.

Fig. 7 summarizes the FatVAP drivers' actions when switching from an *old* interface-AP pair to a *new* pair.

- First, FatVAP traps all future packets handed down by the kernel that need to go out to the old AP and buffers them until the next time this interface-AP pair

is connected.

- Second, FatVAP sends out an 802.11 management frame indicating to the old AP that the host is going into power save mode. The AP then buffers all future packets that need to go to the host.
- Unfortunately, these above two cases do not cover packets that may already be on-the-way, i.e., packets might be in the card's transmit queue waiting to be sent or might even be in the air. To prevent packet loss, FatVAP waits a little bit for the current packet on the air to be received before halting the hardware. FatVAP also preserves the packets waiting in the interface's transmit queue. The transmit queue of the old interface is simply detached from the HAL and is re-attached when the interface is next scheduled.
- Fourth, FatVAP resets the hardware settings of the card and pushes the new association state into the HAL. If the new AP is on a different channel, the card changes channels and listens at the new frequency band.
- Finally, waking up the new interface is simple as the hardware is now on the right channel. FatVAP sends out a management frame telling the new AP that the host is coming out of power save, the AP immediately starts forwarding buffered packets to the host.

3.3.2 Transparent Switching

We would like FatVAP to work with unmodified APs. Switching APs transparently involves handling these practical deployment scenarios.

(a) Cannot Use a Single MAC Address: When two APs are on the same 802.11 channel (operate in the same frequency band), as in Fig. 8a, you cannot connect to both APs with virtual interfaces that have the same MAC address. To see why this is the case, suppose the client uses both AP1 and AP2 that are on the same 802.11 channel. While exchanging packets with AP2, the client claims to AP1 that it has gone into the power-save mode. Unfortunately, AP1 overhears the client talking to AP2 as it is on the same channel, concludes that the client is out of power-save mode, tries to send the client its buffered packets and when unsuccessful, forcefully deauthenticates the client.

FatVAP confronts MAC address problems with an existing feature in many wireless chipsets that allows a physical card to have multiple MAC addresses [4]. The trick is to change a few of the most significant bits across these addresses so that the hardware can efficiently listen for packets on all addresses. But, of course, the number of such MAC addresses that a card can fake is limited. Since the same MAC address can be reused for APs that are on different channels, FatVAP creates a pool of interfaces, half of which have the primary MAC, and the rest have

unique MACs. When FatVAP assigns a MAC address to a virtual interface, it ensures that interfaces connected to APs on the same channel do not share the MAC address.

(b) Light-Weight APs (LWAP): Some vendors allow a physical AP to pretend to be multiple APs with different ESSIDs and different MAC addresses that listen on the same channel, as shown in Fig. 8b. This feature is often used to provide different levels of security (e.g., one light-weight AP uses WEP keys and the other is open) and traffic engineering (e.g., preferentially treat authenticated traffic). For our purpose of aggregating AP bandwidth, switching between light weight APs is useless as the two APs are physically one AP.

FatVAP uses a heuristic to identify light-weight APs. LWAPs that are actually the same physical AP share many bits in their MAC addresses. FatVAP connects to only one AP from any set of APs that have fewer than five bits different in their MAC addresses.

4 EVALUATION

We evaluate our implementation of FatVAP in the Madwifi driver in an internal testbed we built with APs from Cisco and Netgear, in hotspots served by commercial providers like T-Mobile, and in residential areas which have low-cost APs connected to DSL or cable modem backends.

Our results reveal three main findings.

- In the testbed, FatVAP performs as expected. It balances load across APs and aggregates their available backhaul bandwidth, limited only by the wireless capacity and application demands. This result is achieved even when the APs are on different wireless channels.
- In today's residential and Hotspot deployments (in Cambridge/Somerville, MA), FatVAP delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.
- FatVAP safely co-exists with unmodified drivers and other FatVAP clients. At each AP, FatVAP competes with unmodified clients as fairly as an unmodified MadWifi driver, and is sometimes fairer because FatVAP moves away from congested APs. FatVAP clients are also fair among themselves.

4.1 Experimental Setup

(a) Drivers We compare the following two drivers.

- *Unmodified Driver:* This refers to the madwifi v0.9.3 [4] driver. On linux, MadWifi is the current *defacto* driver for Atheros chipsets and is a natural baseline.
- *FatVAP:* This is our implementation of FatVAP as an extension of madwifi v0.9.3. Our implementation includes the features described in §3, and works in con-

Operation	Time (μ s)	
	Mean	STD
IP Checksum Recompute	0.10	0.09
TCP/UDP Checksum Recompute	0.12	0.14
Flow Lookup/Add in HashTable	2.52	2.30
Running the Scheduler	16.21	4.85
Switching Delay	2897.48	2780.71

Table 1: Latency overhead of various FatVAP operations.

junction with autorate algorithms, carrier-sense, CTS-to-self protection, etc.

(b) Access Points: Our testbed uses Cisco Aironet 1130AG Series access points and Netgear’s lower-cost APs. We put the testbed APs in the 802.11a band so as to not interfere with our lab’s existing infrastructure. Our outside experiments run in hotspots and residential deployments and involve a variety of commercial APs in the 802.11b/g mode, which shows that FatVAP works across 802.11a/b/g. The testbed APs can buffer up to 200 KB for a client that enters the power-save mode.³ Testbed APs are assigned different 802.11a channels (we use channels 40, 44, 48, 52, 56 and 60). The wireless throughput to all APs in our testbed is in the range [19 – 22] Mb/s. The actual value depends on the AP, and differs slightly between uplink and downlink scenarios. APs in hotspots and residential experiments have their own channel assignment which we do not control.

(c) Wireless Clients: We have tested with a few different wireless cards, from the Atheros chipsets in the latest Thinkpads (Atheros AR5006EX) to older Dlink and Netgear cards. Clients are 2GHz x86 machines that run Linux v2.6. In each experiment, we make sure that FatVAP and the compared unmodified driver use similar machines with the same kernel version/revision and the same card.

(d) Traffic Shaping: To emulate an AP backhaul link, we add a traffic shaper behind each of our test-bed APs. This shaper is a Linux box that bridges the APs traffic to the Internet and has two Ethernet cards, one of which is plugged into the lab’s (wired) GigE infrastructure, and the other is connected to the AP. The shaper controls the end-to-end bandwidth through a token bucket based rate-filter whose rate determines the capacity of AP’s access link. We use the same access capacity for both downlink and uplink.

(e) Traffic Load: All of our experiments use TCP. A FatVAP client assigns traffic to APs at the granularity of a TCP flow as described in §3.2. An unmodified client assigns traffic to the single AP chosen by its unmodified driver [4]. Each experiment uses one these traffic loads.

³We estimate this value by computing the maximum burst size that a client obtains when it re-connects after spending a long time in the power-save mode.

- *Long-lived iperf TCP flows:* In this traffic load, each client has as many parallel TCP flows as there are APs. Flows are generated using iperf [2] and each flow lasts for 5 minutes.
- *Web Traffic:* This traffic load mimics a user browsing the Web. The client runs our modified version of WebStone 2.5 [8] a tool for benchmarking Web servers. Requests for new Web pages arrive as a Poisson process with a mean of 2 pages/s, the number of objects on a page is exponentially distributed with a mean of 20 objects/page, the objects themselves are copies of actual content on the CSAIL Web server and have sizes that are roughly a power-law with mean equal to 15KB. Note that popular browsers usually open multiple parallel connections to the same server or different servers to quickly download the various objects on a web page (e.g., images, scripts) [18].
- *BitTorrent:* Here, we use the Azureus [1] BitTorrent client to fetch a 500MB file. The tracker is on a CSAIL machine, and 8 Planetlab nodes act as peers. Note that BitTorrent fetches data in parallel from multiple peers.

4.2 Microbenchmarks

To profile the various components of FatVAP, we use the x86 rdtsc instruction for fine-grained timing information. rdtsc reads a hardware timestamp counter that is incremented once every CPU cycle. On our 2 GHz client, this yields a resolution of 0.5 nano seconds.

Table 1 shows our microbenchmarks. The table shows that the delay seen by packets on the fast-path (e.g., flow lookup to find which AP the packets need to go to, re-computing checksums) is negligible. Similarly, the overhead of computing and updating the scheduler is minimal. The bulk of the overhead is caused by AP switching. It takes an average of 3 ms to switch from one AP to another. This time includes sending a power save frame, waiting until the HAL has finished sending/receiving the current packet, switching both the transmit and receive queues, switching channel/AP, and sending a management frame to the new AP informing it that the client is back from power save mode. The standard deviation is also about 3 ms, owing to the variable amount of pending interrupts that have to be picked up. Because FatVAP performs AP switching in the driver, its average switching delay is much lower than prior systems (3ms as opposed to 30-600ms). We note that switching cost directly affects the throughput a user can get. A user switching between two APs every 100ms, would only have 40ms of usable time left if each switch takes 30ms, as opposed to 94ms of usable time when each switch takes 3ms and hence can more than double his throughput (94% vs. 40% use).

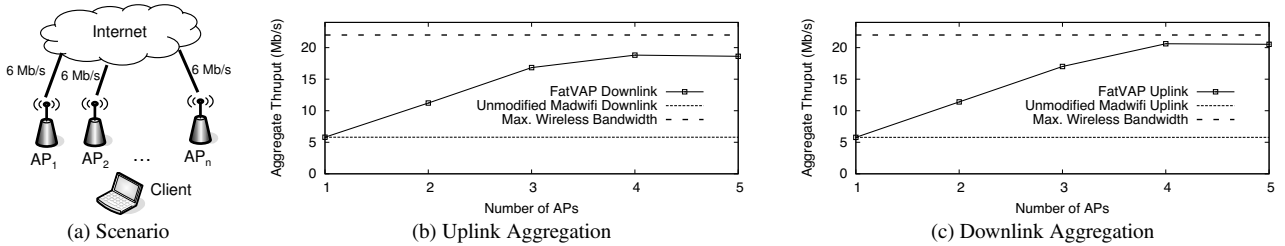


Figure 9: FatVAP aggregates AP backhaul bandwidth until the limit of the card's wireless bandwidth, i.e., wireless link capacity — switching overhead.

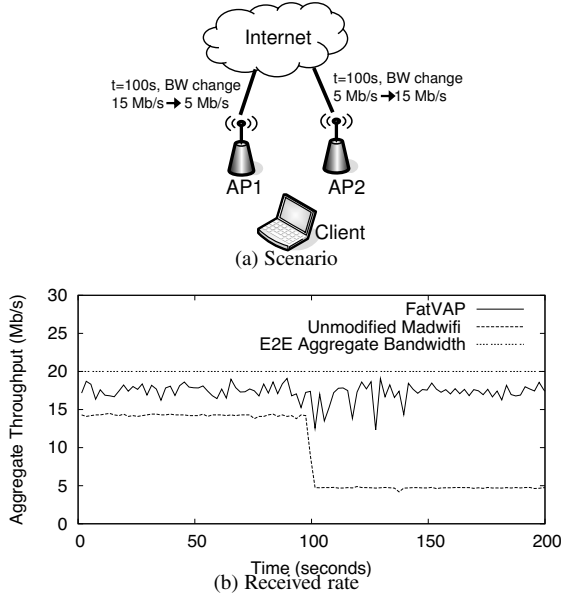


Figure 10: At time $t = 100$ s, the available bandwidth at the first access link changes from 15Mb/s to 5Mb/s, whereas the available bandwidth at the second access link changes from 5Mb/s to 15Mb/s. FatVAP quickly rebalances the load and continues to deliver the sum of the APs' available end-to-end bandwidth. In the scenario, an unmodified driver limits the client to AP1's available bandwidth.

4.3 Can FatVAP Aggregate AP Backhaul Rates?

FatVAP's main goal is to allow users in a hotspot or at home to aggregate the bandwidth available at all accessible APs. Thus, in this section we check whether FatVAP can achieve this goal.

Our experimental setup shown in Fig. 9(a) has $n = \{1, 2, 3, 4, 5\}$ APs. The APs are on different channels and each AP has a relatively thin access link to the Internet (capacity 6Mb/s), which we emulate using the traffic shaper described in §4.1(c). The wireless bandwidth to the APs is about $[20 - 22]$ Mb/s. The traffic constitutes of long-lived iperf TCP flows, and there are as many TCPs as APs, as described in §4.1(d). Each experiment is first performed by FatVAP, then repeated with an unmodified driver. We perform 20 runs and compute the average throughput across them. The question we ask is: does FatVAP present its client with a fat virtual AP, whose backhaul bandwidth is the sum of the AP's backhaul bandwidths?

Figs. 9(b) and 9(c) show the aggregate throughput of the FatVAP client both on the uplink and downlink, as a

function of the number of APs that FatVAP is allowed to access. When FatVAP is limited to a single AP, the TCP throughput is similar to running the same experiment with an unmodified client. Both throughputs are about 5.8Mb/s, slightly less than the access capacity because of TCP's sawtooth behavior. But as FatVAP is given access to more APs, its throughput doubles, and triples. At 3 APs, FatVAP's throughput is 3 times larger than the throughput of the unmodified driver. As the number of APs increases further, we start hitting the maximum wireless bandwidth, which is about 20-22Mb/s. Note that FatVAP's throughput stays slightly less than the maximum wireless bandwidth due to the time lost in switching between APs. FatVAP achieves its maximum throughput when it uses 4 APs. In fact, as a consequence of switching overhead, FatVAP chooses not to use the fifth AP even when allowed access to it. Thus, one can conclude that FatVAP effectively aggregates AP backhaul bandwidth up to the limitation imposed by the maximum wireless bandwidth.

4.4 Can FatVAP Adapt to Changes in Bandwidth?

Next, if an AP's available bandwidth changes, we would like FatVAP to re-adjust and continue delivering the sum of the bandwidths available across all APs. Note that an unmodified MadWifi cannot respond to changes in backhaul capacity. On the other hand, FatVAP's constant estimation of both end-to-end and wireless bandwidth allows it to react to changes within a couple of seconds. We demonstrate this with the experiment in Fig. 10, where two APs are bottlenecked at their access links. As before, the APs are on two different channels, and the bandwidth of the wireless links to the APs is about $[21-22]$ Mb/s. At the beginning, AP1 has 15Mb/s of available bandwidth, whereas AP2 has only 5Mb/s. At time $t = 100$ s, we change the available bandwidth at the two APs, such that AP1 has only 5Mb/s and AP2 has 15 Mb/s. Note that since the aggregated available bandwidth remains the same, FatVAP should deliver constant throughput across this change. We perform the experiment with a FatVAP client, and repeat it with an unmodified client that connects to AP1 all the time. In both cases, the client uses iperf [2] to generate large TCP flows, as described in §4.1(d).

Fig. 10(b) shows the client throughput, averaged over

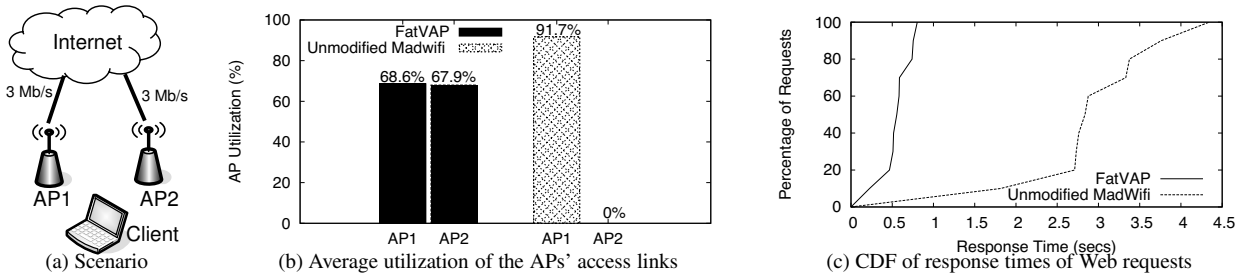


Figure 11: Because it balances the load across the two APs, FatVAP achieves a significantly lower response time for Web traffic in comparison with an unmodified driver.

2s intervals, as a function of time. The figure shows that if the client uses an unmodified driver connected to AP1, its throughput will change from 15Mb/s to 5Mb/s in accordance with the change in the available bandwidth on that AP. FatVAP, however, achieves a throughput of about 18 Mb/s, and is limited by the sum of the APs' access capacities rather than the access capacity of a single AP. FatVAP also adapts to changes in AP available bandwidth, and maintains its high throughput across such changes.

4.5 Does FatVAP Balance the Load across APs?

A second motivation in designing FatVAP is to balance load among nearby APs. To check that FatVAP indeed balances AP load, we experiment with two APs and one client, as shown in Fig. 11(a). We emulate a user browsing the Web. Web sessions are generated using WebStone 2.5, a benchmarking tool for Web servers [8] and fetch Web pages from a Web server that mirrors our CSAIL Web-server, as described in §4.1(d).

Fig. 11(b) shows that FatVAP effectively balances the utilization of the APs' access links, whereas the unmodified driver uses only one AP, congesting its access link. Fig. 11(c) shows the corresponding response times. It shows that FatVAP's ability to balance the load across APs directly translates to lower response times for Web requests. While the unmodified driver congests the default APs causing long queues, packet drops, TCP time-outs, and thus long response times, FatVAP balances AP loads to within a few percent, preventing congestion, and resulting in much shorter response times.

4.6 Does FatVAP Compete Fairly with Unmodified Drivers?

We would like to confirm that regardless of how FatVAP schedules APs, a competing unmodified driver would get its fair share of bandwidth at an AP. We run the experiment in Fig. 12(a), where a FatVAP client switches between AP1 and AP2, and shares AP1 with an unmodified driver. In each run, both clients use iperf to generate long-lived TCP flows, as described in §4.1(d). For the topology in Fig. 12(a), since AP1 is shared by two clients, we have $w_1 = 19/2 = 9.5\text{Mb/s}$, and $e_1 = 10/2 =$

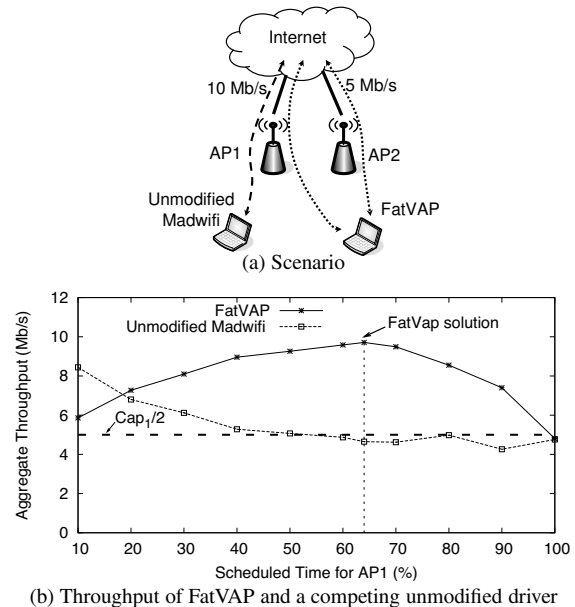


Figure 12: FatVAP shares the bandwidth of AP1 fairly with the unmodified driver. Regardless of how much time FatVAP connects to AP1, the unmodified driver gets half of AP1's capacity, and sometimes more. The results are for the downlink. The uplink shows a similar behavior.

5Mb/s. AP2 is not shared, and has $w_2 = 20\text{Mb/s}$, and $e_2 = 5\text{Mb/s}$.

Fig. 12(b) plots the throughput of both FatVAP and an unmodified driver when we impose different time-sharing schedules on FatVAP. For reference, we also plot a horizontal line at 5Mb/s, which is one half of AP1's access capacity. The figure shows that regardless of how much time FatVAP connects to AP1, it always stays fair to the unmodified driver, that is, it leaves the unmodified driver about half of AP1's capacity, and sometimes more. FatVAP achieves the best throughput when it spends about 55-70% of its time on AP1. Its throughput peaks when it spends about 64% of its time on AP1, which is, in fact, the solution computed by our scheduler in §3.1 for the above bandwidth values. This shows that our AP scheduler is effective in maximizing client throughput.

4.7 Are FatVAP Clients Fair Among Themselves?

When unmodified clients access multiple APs the aggregate bandwidth is divided at the coarse granularity of a

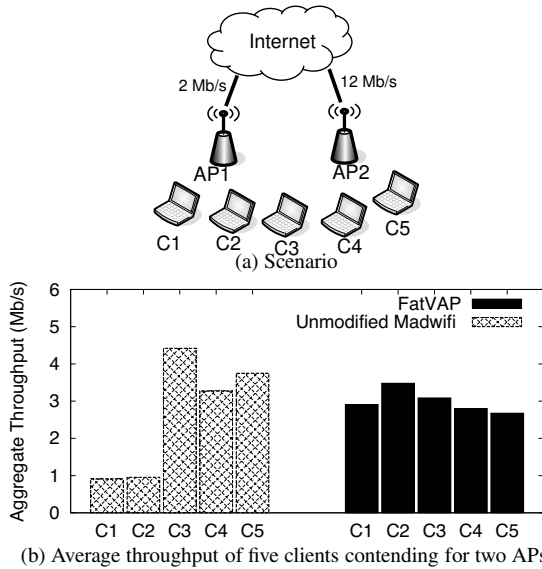


Figure 13: FatVAP clients compete fairly among themselves and have a fairer throughput allocation than unmodified clients under the same conditions.

client. This causes significant unfairness between clients that use different APs. The situation is further aggravated since unmodified clients pick APs based on signal strength rather than available bandwidth, and hence can significantly overload an AP.

Here, we look at 5 clients that compete for two APs, where AP1 has 2Mb/s of available bandwidth and AP2 has 12 Mb/s, as shown in Fig. 13(a). The traffic load consists of long-lived iperf TCP flows, as described in §4.1(d). Fig. 13(b) plots the average throughput of clients with and without FatVAP. With an unmodified driver, clients C1 and C2 associate with AP1, thereby achieving a throughput of less than 1Mb/s. The remaining clients associate with AP2 for roughly 4Mb/s throughput for each. However, FatVAP’s load balancing and fine-grained scheduling allow all five clients to fairly share the aggregate bandwidth of 14 Mb/s, obtaining a throughput of roughly 2.8 Mb/s each, as shown by the dark bars in Fig. 13(b).

4.8 FatVAP in Residential Deployments

We demonstrate that FatVAP can bring real and immediate benefits in today’s residential deployments. To do so, we experiment with FatVAP in three residential locations in Cambridge, MA, shown in Fig. 14. Each of these locations has two APs, and all of them are homes of MIT students, where neighbors are interested in combining the bandwidth of their DSL lines. Again, in these experiments, we run Web sessions that access a mirror of the CSAIL Web-server, as explained in §4.1(d). In each location, we issue Web requests for 10 min, and repeat the experiment with and without FatVAP.

Fig. 15(a) plots the CDF of throughput taken over all

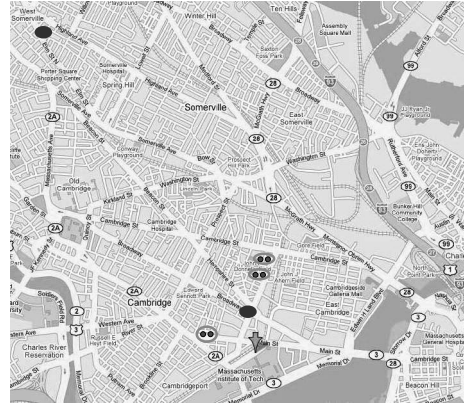


Figure 14: Location of residential deployments (in red) and hotspots (in blue)

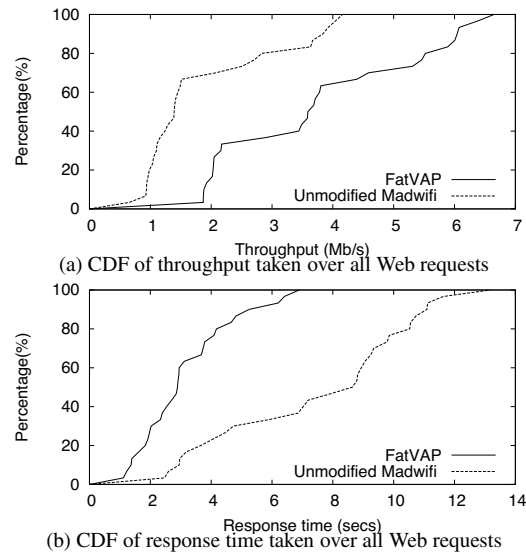


Figure 15: FatVAP’s performance in three residential deployments in Cambridge, MA. The figure shows that FatVAP improves the median throughput by 2.6x and reduces the median response time by 2.8x.

the web requests in all three locations. The figure shows that FatVAP increases the median throughput in these residential deployments by 2.6x. Fig. 15(b) plots the CDF of the response time taken over all requests. The figure shows that FatVAP reduces the median response time by 2.8x. Note that though all these locations have only two APs, Web performance more than doubled. This is due to FatVAP’s ability to balance the load across APs. Specifically, most Web flows are short lived and have relatively small TCP congestion windows. Without load balancing, the bottleneck drops a large number of packets, causing these flows to time out, which results in worse throughputs and response times. In short, our results show that FatVAP brings immediate benefit in today’s deployments, improving both client’s throughput and response time.

4.9 Hotspots

Results in Hotspots show that FatVAP can aggregate throughput across commercial access points. The traf-

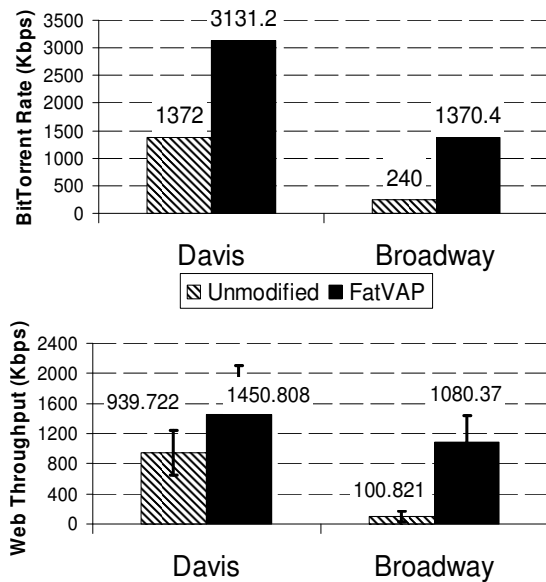


Figure 16: FatVAP's performance in two hotspots in Cambridge/Somerville, MA, showing that FatVAP improves throughput for Web downloads and BitTorrent.

fic load uses both Web downloads and BitTorrent and is generated as described in §4.1(d). Our results show that both Web sessions and BitTorrent obtain much improved throughput compared to the unmodified driver. Fig. 16 shows that, depending on the Hotspot, FatVAP delivers an average throughput gain of 1.5-10x to Web traffic, and 2 – 6x to BitTorrent. The huge gains obtained in the Broadway site are because the AP with the highest RSSI was misconfigured with a very large queue size. When congested, TCPs at this AP experienced a huge RTT inflation, time-outs, and poor throughput.

5 RELATED WORK

Related work falls in two main areas.

(a) Connecting to Multiple APs: There has been much interest in connecting a wireless user to multiple networks. Most prior work uses separate cards to connect to different APs or cellular base stations [5, 23, 27]. PERM [27] connects multiple WiFi cards to different residential ISPs, probes the latency via each ISP, and assigns flows to cards to minimize latency. Horde [23] uses multiple cellular connections via different providers. In contrast to this work which stripes traffic across independent connections, FatVAP uses the same card to associate and exchange data with multiple APs. Further, FatVAP uses virtual connections to these APs that are very much dependent and so are the throughput estimates that FatVAP uses to choose APs.

The closest to our work is the MultiNet project [13], which was later named VirtualWiFi [6]. MultiNet abstracts a single WLAN card to appear as multiple virtual WLAN cards to the user. The user can then configure each virtual card to connect to a different wireless

network. MultiNet applies this idea to extend the reach of APs to far-away clients and to debug poor connectivity. We build on this vision of MultiNet but differ in design and applicability. First, MultiNet provides switching capabilities but says nothing about which APs a client should toggle and how long it should remain connected to an AP to maximize its throughput. In contrast, FatVAP schedules AP switching to maximize throughput and balance load. Second, FatVAP can switch APs at a fine time scale and without dropping packets; this makes it the only system that maintains concurrent TCP connections on multiple APs.

(b) AP Selection: Current drivers select an AP based on signal strength. Prior research has proposed picking an AP based on load [20], potential bandwidth [28], and a combination of metrics [21]. FatVAP fundamentally differs from these techniques in that it does not pick a single AP, but rather multiplexes the various APs in a manner that maximizes client throughput.

6 DISCUSSION

Here, we discuss some related issues and future work.

(a) Multiple WiFi Cards: While FatVAP benefits from having multiple WiFi cards on the client's machine, it does not rely on their existence. We made this design decision for various reasons. First most wireless equipments naturally come with one card and some small handheld devices cannot support multiple cards. Second, without FatVAP the number of cards equals the number of APs that one can connect with, which limits such a solution to a couple of APs. Third, cards that are placed very close to each other may interfere; WiFi channels overlap in their frequency masks [16] and could leak power to each other's bands particularly if the antennas are placed very close. Forth, even with multiple cards, the client still needs to pick which APs to connect to and route traffic over these APs as to balance the load. FatVAP does not constrain the client to having multiple cards. If the client however happens to have multiple cards, FatVAP would allow the user to exploit this capability to expand the number of APs that it switches between and hence improve the overall throughput.

(b) Channel Bonding and Wider Bands: Advances like channel bonding (802.11n) and wider bands (40MHz wide channels) increase wireless link capacity to hundreds of Mb/s. Such schemes widen the gap between the capacity of the wireless link and the AP's backhaul link, making FatVAP more useful. In such settings, FatVAP lets one wireless card collect bandwidth from tens of APs.

(c) WEP and Splash Screens: We are in the process of adding WEP and splash-screen login support to our FatVAP prototype. Supporting WEP keys is relatively easy, the user needs to provide a WEP key for every secure AP

that he wants to access. FatVAP reads a pool of known $\langle \text{WEP key, ESSID} \rangle$ tuples and uses the right key to talk to each protected AP. Supporting splash screen logins used by some commercial hot-spots, is a bit more complex. One would need to pin all traffic to an AP for the duration of authentication, after which FatVAP can distribute traffic as usual.

7 CONCLUSION

Prior work has documented the abundance of 802.11 access points and the fact that APs occur in bunches—if you see one, it is likely that many others are close by. This paper takes the next step by aggregating bandwidth across the many available APs, that may be spread across different channels. To the best of our knowledge, FatVAP is the first driver to choose how long to connect to each AP, maintain concurrent TCP flows through multiple APs and provide increased throughput to unmodified applications. FatVAP requires no changes to the 802.11 MAC or to access points. Fundamentally, FatVAP relaxes the constraint that a user with one card can only connect with one access point to achieve both better performance for users and a better distribution of load across APs.

ACKNOWLEDGMENTS

We thank our shepherd Srinu Seshan, Ranveer Chandra, Szymon Chachulski, Nate Kushman, Hariharan Rahul, Sachin Katti and Hari Balakrishnan for comments that improved this paper. This work is supported by the NSF Career Award CNS-0448287 and an ITRI gift. The opinions and findings in this paper are those of the authors and do not necessarily reflect the views of NSF or ITRI.

REFERENCES

- [1] Azureus. <http://azureus.sourceforge.net>.
- [2] iperf. <http://dast.nlanr.net/Projects/Iperf>.
- [3] Knapsack. http://en.wikipedia.org/wiki/Knapsack_problem.
- [4] Madwifi. <http://madwifi.org>.
- [5] Slurpr. <http://geektechnique.org/projectlab/781/slurpr-the-mother-of-all-wardrive-boxes>.
- [6] Virtual WiFi. <http://research.microsoft.com/netres/projects/virtualwifi>.
- [7] Virtual WiFi-FAQ. <http://research.microsoft.com/netres/projects/virtualwifi/faq.htm>.
- [8] WebStone. <http://www.mindcraft.com/webstone/>.
- [9] WiFi411. <http://www.wifi411.com>.
- [10] Wigle.net. <http://wigle.net>.
- [11] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks. In *MobiCom*, 2004.
- [12] P. Bahl, R. Chandra, T. Moscibroda, Y. Wu, and Y. Yuan. Load Aware Channel-Width Assignments in Wireless LANs. Technical Report TR-2007-79, Microsoft Research, 2007.
- [13] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In *INFOCOM*, 2004.
- [14] K. Egevang and P. Francis. The IP Network Address Translator (NAT). In *RFC 1631*, May 1994.
- [15] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control. In *SIGCOMM*, 1999.
- [16] M. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly, 2005.
- [17] M. Jain and C. Dovrolis. Pathload: A Measurement Tool for End-to-End Available Bandwidth. In *PAM*, March 2002.
- [18] H. Jamjoom and K. G. Shin. Persistent Dropping: An Efficient Control of Traffic. In *ACM SIGCOMM*, 2003.
- [19] H. Jiang and C. Dovrolis. Passive Estimation of TCP Round-Trip Times. In *ACM CCR*, 2002.
- [20] G. Judd and P. Steenkiste. Fixing 802.11 access point selection. *ACM CCR*, 2002.
- [21] A. J. Nicholson, Y. Chawathe, M. Y. Chen, B. D. Noble, and D. Wetherall. Improved Access Point Selection. In *MobiSys*, 2006.
- [22] K. Papagiannaki, N. Taft, and C. Diot. Impact of Flow Dynamics on Traffic Engineering Design Principles. In *INFOCOM*, 2004.
- [23] A. Qureshi and J. Guttag. Horde: Separating Network Striping Policy from Mechanism. In *MobiSys*, 2005.
- [24] M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, and J. Zahorjan. Measurement-based Characterization of 802.11 in a Hotspot Setting. In *SIGCOMM E-WIND Workshop*, 2005.
- [25] S. Kandula and D. Katabi and S. Sinha and A. Berger. Dynamic Load Balancing Without Packet Reordering. In *CCR*, 2006.
- [26] J. Strauss, D. Katabi, and F. Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *IMC*, Oct. 2003.
- [27] N. Thompson, G. He, and H. Luo. Flow Scheduling for End-host Multihoming. In *IEEE INFOCOM*, 2006.
- [28] S. Vasudevan, D. Papagiannaki, and C. Diot. Facilitating Access Point Selection in IEEE 802.11 Wireless Networks. In *IMC*, 2005.

A END-TO-END AVAILABLE RATE INFLATION

Suppose that the client spends f_i of its duty cycle D at an AP that has wireless and end-to-end available bandwidths, w_i and e_i . Thus, the client spends $f_i \times D$ time at this AP and the remaining $(1 - f_i)D$ time at other APs. The AP buffers data that it receives when the client is away and delivers this data when the client next connects to the AP. Let x be the amount served out of the AP's buffer, then

$$x = e_i \left((1 - f_i)D + \frac{x}{w_i} \right) \quad (14)$$

Eq. 14 means that the buffer gets data at rate e_i during two phases: when the client is away from the AP and when data in the buffer is being delivered to the client. The first phase lasts for $(1 - f_i)D$ and the second lasts for $\frac{x}{w_i}$. The total data received by the client in D seconds is,

$$\text{DataReceived} = x + e_i(f_i D - \frac{x}{w_i}). \quad (15)$$

This means simply that the client receives x units from the buffer in time $\frac{x}{w_i}$ and once the buffer is depleted, receives data at the end-to-end rate e_i for the remaining $f_i D - \frac{x}{w_i}$. Since the client listens at the AP for

$$\text{ListenTime} = f_i D, \quad (16)$$

the client's estimate of end-to-end available bandwidth is

$$\text{Estimate} = \frac{\text{DataReceived}}{\text{ListenTime}} = \frac{e_i}{f_i}. \quad (17)$$

Eq 17 is obtained by eliminating x from Eqs. 14, 15, 16. But, if the fraction of time that the client listens to the AP is smaller than $\frac{e_i}{w_i}$, it is easy to see that the client will always be served data from the AP buffer at the wireless available bandwidth w_i . Hence, the estimate is

$$\text{Estimate} = \max \left(\frac{e_i}{f_i}, w_i \right). \quad (18)$$

Efficiency through Eavesdropping: Link-layer Packet Caching

Mikhail Afanasyev, David G. Andersen[†], and Alex C. Snoeren
University of California, San Diego and [†]Carnegie Mellon University
{mafanasyev,snoeren}@cs.ucsd.edu, dga@cs.cmu.edu

Abstract

The broadcast nature of wireless networks is the source of both their utility and much of their complexity. To turn what would otherwise be unwanted interference into an advantage, this paper examines an entirely backwards-compatible extension to the 802.11 link-layer protocol for making use of overheard packets, called RTS-id. RTS-id operates by augmenting the standard 802.11 RTS/CTS process with a packet ID check, so that if the receiver of an RTS message has already received the packet in question, it can inform the sender and bypass the data transmission entirely.

We present the design, implementation, and evaluation of RTS-id on a real hardware platform that provides a DSP-programmable 802.11 radio. While limited in scale due to restricted availability of the CalRadio platform, our testbed experiments demonstrate that RTS-id can reduce air time usage by 25.2% in simple 802.11b infrastructure deployments on real hardware, even when taking into account all of the protocol overhead. Additionally, we present trace-based simulations demonstrating the potential savings on the MIT Roofnet mesh network. In particular, RTS-id provides a 12% decrease in the number of expected data transmissions for a median path, and over 25% reduction for more than 10% of Roofnet paths.

1 Introduction

Multi-hop wireless networks are becoming a popular mechanism for providing Internet access, both in urban areas [5] and in rural and developing settings [17]. By reducing the need for a fixed wired infrastructure, they offer the hope of providing cheaper connectivity and faster deployment. These networks, however, face a number of challenges not shared by their wired counterparts: interference, multi-path losses, and rapidly changing, unpredictable connectivity patterns. Wireless networks are by nature *broadcast*: a transmission from one node may interfere with or be received by multiple other nodes. The broadcast nature of these networks and the requirement

that nodes forward traffic on behalf of one another is one of the primary scaling limitations of multi-hop wireless networks [16].

The question we ask in this paper is: how can we turn the broadcast nature of wireless to our advantage, instead of leaving it purely an interference-causing liability. Past approaches to doing so include caching opportunistically overheard objects (e.g., in satellite-based distribution systems [2]); by modifying ad hoc routing protocols to enable them to acknowledge packets received later in the forwarding chain [6]; by using network coding on bi-directional traffic streams [14]; and, most recently, by using network coding to achieve similar benefits without explicit coordination [9]. We examine instead a simple per-hop link-layer modification, that we call *RTS-id*, that takes advantage of overheard packets in a protocol and topology-independent manner that requires only the co-operation of adjacent nodes in a path.

While previous systems like ExOR and MORE have demonstrated dramatic performance improvements, ranging up to throughput factors of 2–10 [6, 9], it is important to note that these improvements do *not* apply to interactive and highly asymmetric TCP connections common in access networks. Even the most recent opportunistic routing techniques in MORE require batching packets together for transmission, which interacts poorly with TCP's congestion control. Furthermore, to realize these gains, the entire network must be upgraded to support the enhanced routing and forwarding architecture. In contrast, RTS-id is backwards compatible with existing 802.11 hardware: individual nodes can be upgraded by replacing the 802.11 driver and/or firmware, yet they will continue to interoperate with legacy nodes. We verify that the RTS-id extensions are ignored by hardware that does not support it with no ill effects. Furthermore, while substantially more modest than the bulk transfer improvements demonstrated by other systems, the gains we report are independent of transport-layer protocol: they are equally applicable to UDP and TCP. Finally, we demonstrate that significant gains can be had in typical infrastructure deployments as well.

The key property that we build upon is that wireless delivery is probabilistic. Two nearby nodes may hear 90% of each others' transmissions, and two nodes farther away may hear only 20%. This highly variable, unreliable packet reception is the critical challenge to the design of practical ad hoc routing protocols; to minimize metrics such as the expected transmission count [11] or expected transmission time [4] protocols will often select paths in which there is a non-zero chance that packets could be overheard farther along the forwarding chain.

Like previous schemes [14], we argue that all nodes should optimistically cache recently received packets regardless of their destination. Rather than require additional coordination, however, we propose to extend the normal 802.11 RTS/CTS exchange to include a packet ID. If a receiver already has a packet cached, it can respond to an RTS directly with an ACK. Our RTS-id mechanism has the potential to elegantly optimize at least three distinct common occurrences in wireless 802.11 networks:

- **“Node-to-node via access point”.** When two nodes on the same wireless LAN communicate, they *must* do so through the AP(s) to which they are associated, even if they can directly hear each others communication. RTS-id allows the AP to avoid relaying packets to the receiver if the receiver heard the initial transmission.
- **“Hop-over” transmission.** More generally, when sending packets through a series of nodes $A \rightarrow B \rightarrow C$, the probability of C hearing A 's initial broadcast is often non-zero. When B then attempts to relay the packet to C , RTS-id allows it to immediately respond “I already have this packet.”
- **Packet retransmissions.** If a link-layer ACK is lost, RTS-id prevents spurious retransmissions by short-circuiting the retransmission.

The key contributions of the current work are the design of RTS-id and a backwards-compatible implementation on a real 802.11 hardware platform, CalRadio. We show that RTS-id can decrease air time usage by 25.2% when two nodes communicate to each other through an access point in our testbed. Moreover, we demonstrate via simulation the potential performance improvement in a large-scale multi-hop deployment. Using publicly available data from the MIT Roofnet community network, we show that RTS-id can decrease by 12% the expected number of data packet transmissions for the median route when compared to Roofnet's existing routing protocol, and by over 40% for the most-improved routes. Even if a network does not normally use RTS/CTS, RTS-id decreases by up to 15% the air time for fully optimized Roofnet routes after accounting for the additional overhead (and without considering any potential benefits from

avoided hidden terminal collisions). Perhaps most importantly, we show that RTS-id can significantly enhance the performance of much simpler routing algorithms—so much so that they out-perform Roofnet's complex routing algorithm that requires maintaining accurate, up-to-date loss information about all node pairs.

The remainder of this paper is organized as follows. We begin in Section 2 with an overview of previous schemes that leverage overhearing. Section 3 presents the design of RTS-id, and we describe our prototype CalRadio implementation in Section 4. We systematically evaluate the potential performance improvement in both infrastructure networks (Section 5) and the MIT Roofnet mesh network (Section 6) before discussing security concerns (Section 7) and future work in Section 8.

2 Related work

The work philosophically most related to ours is ExOR [6, 7] and the more recent MORE [9]. Both protocols essentially define new, bulk-transfer transport protocols to increase efficiency. ExOR aggressively batches packet transmissions together to take advantage of overheard transmissions. It substantially increases packet throughput, but to do so it requires a redesigned ad hoc routing protocol, and its large batch sizes render ExOR functionally incompatible with traditional transport protocols like TCP and latency-sensitive applications. (The aggressive batching can increase latency by up to 3.5 seconds [6]; its authors acknowledge that “ExOR's batches are likely to interact poorly with TCP's window mechanism” [7].) MORE's operation is similar, but it uses random network coding to avoid the need for ExOR's scheduler. Mostly by increasing opportunities for spatial reuse, MORE achieves unicast throughput 22–45% higher than ExOR's.

In contrast, RTS-id targets the operation of legacy routing and transport protocols, sacrificing some of their impressive performance gains as the price of broader applicability. RTS-id can reduce the number of transmissions required by any existing transport protocol; no changes to applications or operating systems are required.

An alternative technique proposed to harness broadcast is network coding, e.g., COPE [14]. Network coding does not target opportunistic overhearing; rather, it takes advantage of the fact that a sender in the middle of a three-node chain can be heard by both of the nearby nodes during a single transmission, allowing bidirectional traffic to be sent using three transmissions instead of four. Because COPE exploits a different property of broadcast, we believe that its benefits are orthogonal to—and quite possibly compatible with—those from RTS-id. We defer an analysis of this complex interaction for future work. We note, however, that many coding-based approaches

System	Coding	Batch size (packets)	Per-packet overhead	Median improvement
ExOR [6]	none	32	44–114 bytes	60%
MORE [9]	intra-flow	32	<70 bytes	95%
COPE [14]	inter-flow	variable	variable	25–70%
RTS-id	none	1	4 bytes (in RTS)	15–25%*

Table 1: A comparison of mechanisms that leverage the broadcast nature of the wireless channel. Improvement values for ExOR and MORE are taken from the three-node topologies extracted from [9, Figure 6]. COPE numbers reflect TCP and UDP, but *require* bi-directional communication in this simplistic topology. More modest performance gains are possible for uni-directional flows in more complex topologies. *Due to hardware issues, we report savings in terms of air time usage as opposed to throughput; we improve UDP throughput by 26.1% compared to a network with RTS/CTS enabled.

also provide relatively modest gains for the legacy traffic that RTS-id targets, performing optimally only with symmetric UDP traffic. Depending on the degree of asymmetry, the performance improvement may be as low as 5–15%—for UDP. The best case TCP improvement for COPE was 38%, but this is only for networks that do not require RTS/CTS (i.e., do not have any hidden terminals).

Table 1 attempts to summarize the various features of each of these systems. Recognizing that comparing performance claims across implementations and testbeds is problematic, we attempt to give a flavor of the order of magnitude of performance gains offered by each system. We use Roofnet’s routing protocol, Ssrcr, as a baseline, and include the median performance gain over Ssrcr reported by each system’s authors in a three-hop topology. While only MORE and COPE incur coding overhead, the per-packet overhead can be substantial in many cases, rendering the techniques inappropriate for small flows. We were unable to determine the COPE packet header size from [14], but it appears to be of similar order to ExOR and MORE.

A key distinction of RTS-id is its independence from the routing protocol. In particular, we show it can improve the performance of *any* routing protocol—not just Ssrcr. Considerable previous work has examined techniques for selecting routes to leverage opportunistic forwarding opportunities in multi-hop networks [10, 15]. Some prior protocols may be easier to implement than Ssrcr in certain networks; others may be so computationally expensive that it could be more efficient to use simple routes and allow RTS-id to optimize them on-the-fly.

In some ways, RTS-id’s packet cache is reminiscent of the duplicate-suppression cache used in the original DARPA packet radio network [13]. That mechanism lacked RTS-id’s query mechanism, however, only enabling receivers to avoid re-sending packets if they were incorrectly retransmitted or looped.

More generally, Santos and Wetherall proposed a compression mechanism for suppressing long-range packet duplication on wired links [18], later extended to sub-packet duplication by Spring and Wetherall [20]. Unfortunately, these mechanisms do not directly translate to opportunis-

tic wireless: they rely on a near-perfect synchronization of the sender and receivers’ “dictionaries,” exactly the knowledge that does *not* exist in our target environment. However, these techniques suggest promising ways (e.g., combining header compression with RTS-id) for RTS-id to exploit cross-flow and longer-term overhearing. We plan to examine such extensions in future work.

3 RTS-id

Our proposed technique, RTS-id, adds a small exchange *before* packet transmission to ask the receiver if it already has the packet in question. Receivers maintain a small cache of recently observed packets that they check during this exchange. To reduce overhead and ensure backwards-compatibility, RTS-id piggy-backs this query on the existing 802.11 request-to-send (RTS) frames.

RTS/CTS is normally used to reserve the medium for a packet transmission to prevent hidden terminal problems [3]. It operates by having senders broadcast a “request to send” (to a particular receiver) specifying the expected duration of the frame exchange. In accordance with the 802.11 standard, if the receiver determines the channel to be idle, it will reply with a “clear to send” (CTS) frame containing the expected remaining duration of the frame exchange, permitting the sender to begin transmission and informing nearby nodes to remain silent. RTS-id adds a second possibility: the receiver can directly “ACK” the packet with a special CTS-ACK frame.

This section first examines the roles of senders and receivers in RTS-id, then discusses the design alternatives to identify packets. Finally, because RTS-id increases the size of RTS frames (or necessitates their use in a system that does not use them), we discuss how senders and receivers can dynamically enable RTS-id based upon an on-line determination of whether it would benefit them.

3.1 Sender and receiver operation

RTS-id senders operate as shown in Figure 1: they first decide whether to use RTS-id for a packet. If so, they

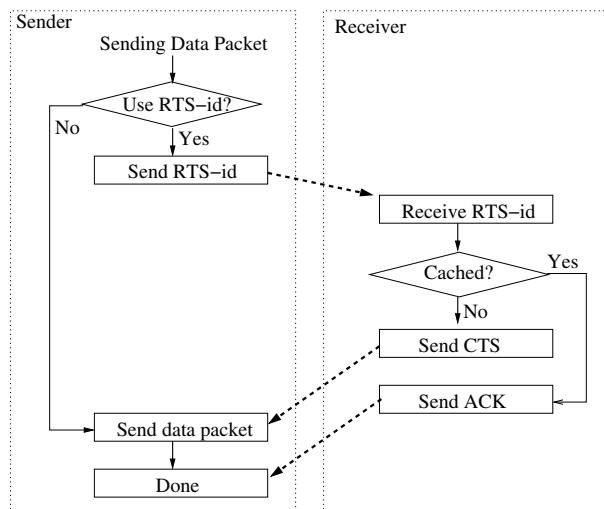


Figure 1: RTS-id operation. For clarity, this figure assumes that the sender does not fall back to normal RTS/CTS use.

transmit an RTS-id frame to the receiver, and expect to receive either a CTS-ACK (the receiver has the packet already) or a normal CTS (the receiver does not have the packet; the sender must transmit). An RTS-id frame is simply a standard RTS frame extended to include a packet ID. With RTS-id, however, rather than setting the duration field to the standard value, it sets it only to the time in microseconds required to transmit the CTS frame and one SIFS (short inter-frame space) interval. This way, nodes overhearing the RTS will only consider the channel reserved for the RTS-id/CTS exchange at this point.

Upon reception of an RTS-id frame, the receiver checks its local packet cache for a packet whose ID matches that in the RTS frame. If present, the receiver sends a CTS-ACK and processes the frame as if it had been received normally. A CTS-ACK is simply a normal CTS frame with the remaining duration field set to zero. This both signals to the sender that the packet was already received, and resets the network allocation vector (NAV) for other stations in the contention domain. If the packet was not found, the receiver sets the CTS duration field to be the same value that would have been used in response to a normal RTS frame, reserving the channel for the time expected to transmit the pending frame, plus one ACK frame and two SIFS intervals.

When a node receives a normal data frame, it operates according to the flowchart in Figure 2. It inserts into a small FIFO cache all received packets larger than `cache_thresh` bytes, regardless of the packet's source or destination. The `cache_thresh` avoids wasting cache entries on small packets such as TCP ACKs. If the packet was previously cached, the receiver informs the sender that the transmission could have been avoided, which enables the adaptive enabling scheme below.

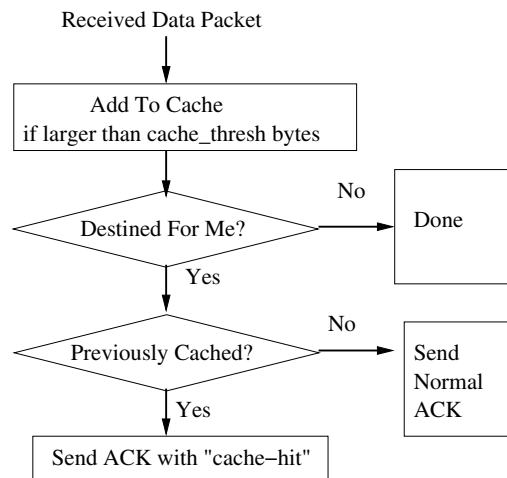


Figure 2: Received packet processing.

3.2 Choice of hash and collisions

RTS-id uses a 32-bit hash of the IP packet contents—not the link-layer frame—as the packet ID. Such a small hash is acceptable if it provides three properties:

Low drop and duplication rate: A hash collision results in both a drop (of the transmitted packet) and a duplication (of the cached packet it collides with). A 32-bit hash with a 64-packet cache will drop about 1 in 67 million packets due to hash collisions. This rate is much lower than typical end-to-end loss on wireless networks.

Independent collisions for transport-layer retransmissions: If the drop probability is non-negligible, then a collision that prevented a particular frame exchange must not cause the end-to-end retransmission of that packet to also be dropped with high probability. This property is provided as long as 1) the hash of the retransmitted packet is different from that of the original; or 2) the contents of the cache differ during the retransmission. Fortunately, both conditions are likely to hold, as several fields in the packet typically change when a packet is retransmitted at the transport or application layers, such as the IP ID, TCP timestamps, DNS query IDs, etc.

Resistant to attacks: The hash should ensure that a non-local attacker cannot guess the ID of a packet and that no attacker can easily craft a packet that will collide with a target packet. We assume that an attacker who transmits on the order of 2^{32} packets over the course of a few seconds has at his command a more effective way of denying service than causing packet collisions. We discuss the security implications of RTS-id, with and without strong hash functions, in Section 7.

3.3 Adaptively enabling RTS-id

RTS-id adds 32 bits of overhead to the small RTS packets. On links in which RTS-id does not provide benefit, this cost may loom large, because 802.11 transmits RTS/CTS packets at the lowest possible rate, 1 Mbps, while the data may be sent at higher rates. Moreover, for networks that would not otherwise use RTS/CTS, the insertion of an entirely new frame exchange comes at considerable cost. Each sender therefore dynamically determines whether or not to use RTS-id when communicating with a particular receiver, based on its past history of cache hits and the size of the packet it is about to transmit.

First, RTS-id processing only considers packets larger than `cache_thresh` ≈ 500 bytes. Smaller packets are transmitted directly (they may, however use normal RTS/CTS depending on the station configuration). For large packets, every participating receiver maintains an RTS-id cache, regardless of whether senders choose to use it. On receiving a packet, the receiver checks its cache to see if the packet had already been heard. If it had, the receiver sets a bit in the ACK packet it sends in response to the packet arrival. Otherwise, it leaves this bit unset. The sender thus is able to determine which packets *would have* resulted in a cache hit had it used RTS-id.¹

On each packet, the sender calculates the (possibly negative) time saved, T_s , by using RTS-id. In the calculation that follows, T_{rtscts} is the time required for a normal RTS-CTS exchange, or zero if RTS-CTS is not enabled.

$$B_s = \text{The bytes saved}$$

$$= \begin{cases} 0 & \text{if no cache hit} \\ \text{Packet size} & \text{if cache hit} \end{cases}$$

$$T_s = \frac{B_s}{rate_{tx}} - (T_{rtsid} - T_{rtscts}).$$

The sender maintains for each (link-level) receiver an exponential weighted moving average with parameter $w \approx 1/200$ of the time saved for each packet:

$$savings = (1 - w) \cdot savings + w \cdot T_s.$$

If the estimated time savings for a particular receiver is large enough, the sender will enable RTS-id. It is not necessary to explicitly enable RTS-id on the receiver: it can promiscuously cache packets whenever sufficient memory and power are available, and may always respond to an RTS-id packet with a normal CTS frame.

4 Implementation

We have implemented RTS-id in a backwards-compatible fashion. RTS-id stations inter-operate seamlessly with

¹To avoid the need to redefine the ACK packet in practice, we overload the “retry” bit. In our experience, current 802.11 devices do not set the retry bit on ACK frames.

RTS:	FC (2)	Dur (2)	Source (6)	Dest (6)	FCS (4)	
RTS-id:	FC (2)	Dur (2)	Source (6)	Dest (6)	FCS (4)	ID (4)

Figure 3: RTS and RTS-id packet formats.

non-RTS-id stations, enabling enhanced performance between adjacent RTS-id capable stations. Our implementation uses the CalRadio 1.0 platform designed and manufactured by CalIT2 [8]. The CalRadio is a low-cost software radio platform consisting of an ARM processor, an on-board Texas Instruments DSP, and a Prism 802.11b baseband processor. The salient feature of the CalRadio for our purposes is that the MAC protocol is implemented almost entirely in C, which allows us to change the format and contents of the RTS and CTS packets. The ARM has access to 4 MB of flash ROM, 2 MB of static RAM and 16 MB of SDRAM, while the DSP operates with 512 KB of RAM. The 802.11 MAC protocol is implemented on the DSP, while the operating system (μ CLinux 2.4.19) and user-level programs run on the ARM.

4.1 Packet details

The RTS-id packet is a simple extension to the standard 802.11 RTS packet as shown in Figure 3. Note that the new ID field is sent *after* the normal RTS frame fields, including the frame check sequence (FCS). Furthermore, when transmitting the RTS-id frame, the length field of the PLCP header is set to the length of the standard RTS frame, *not including* the new ID field. Hence, spec-compliant 802.11 stations that do not support RTS-id will not even decode the hash field, and the frame will look like a normal, well-formed RTS frame.² RTS-id capable stations, however, expecting an RTS-id frame, will know to decode the additional field.

It is important to note that the use of RTS-id does not interfere with the normal ability of RTS/CTS to prevent hidden terminals. The duration specified by the sender’s RTS-id frame will reserve the channel until the end of the RTS-id/CTS exchange. If the data frame is eventually sent, its duration field will update the NAV for all stations in range of the sender. Nodes that hear only the CTS frame will obey its duration field. Because, however, we insert a different value into the RTS-id duration field, the receiver no longer knows how long the pending packet will take to transmit, and is unable to accurately fill out the duration field in the corresponding CTS frame.

To resolve this problem, stations sending a CTS can estimate the appropriate duration based upon a packet size

²The Atheros chip sets we have tested properly decode the RTS-id frame as an RTS. Due to time constraints, we have not yet conducted an exhaustive test of other 802.11 devices. In the worst case, non-compliant stations will simply discard the seemingly mal-formed RTS-id frame with no ill effects.

of `cache_thresh` (smaller packets would not have instigated an RTS-id exchange) and the previous transmission speed used by the sender. (Over-estimating the size prevents hidden terminal problems, but potentially waste air time. Under-estimating creates a small window where a collision may occur that normal RTS/CTS would have prevented.) If greater accuracy is needed, the low-order bits of the RTS-id duration field can be used to encode the approximate size of the pending data packet. Our prototype, however, does not yet implement this extension.

While the ID field is not covered by the FCS (in order to preserve backwards compatibility), a corrupt ID field has little effect. All nodes in our implementation recompute the ID of received packets before insertion into the cache or local delivery, so there is no danger of cache or data corruption. Hence, there are only two issues of concern: First, an ID that should hit in an overhearer's cache is corrupted so that it misses. In this case, an avoidable transmission occurs, resulting in a slight performance decrease. The second, somewhat more expensive case occurs when an ID is corrupted so that it collides with that of a previously overheard case. This situation is no different than a normal hash collision, and occurs (assuming a binary symmetric channel) with equal probability. Such a collision results in a drop (of the corrupted packet) and retransmission (of the packet the ID collided with), impairing performance but not correctness.

4.2 Packet caching and RTS-id

According to the 802.11 specification, a station must respond within 10 microseconds to an RTS request. To interoperate with legacy stations, RTS-id nodes should conform to this response time requirement for both CTS and CTS-ACK packets. We therefore implement the packet cache on the DSP. Due to the tight cycle budget, our implementation uses the CRC32 checksum of invariant [19] packet contents (including the transport layer header and a portion of the payload) as its ID. This choice is obviously deficient with respect to attack resilience; a future implementation will use the low-order 32 bits of a strong cryptographic hash.

4.3 Test-bed deployment

Our current test-bed consists of three CalRadio devices. While CalIT2 distributes CalRadio with basic 802.11b PHY code, the publicly available MAC code is far from complete. We have extended the provided code base to support the core of the 802.11b MAC protocol, including data, ACK, RTS/CTS, and RTS-id/CTS-ACK frames as well as link-layer retransmission and collision avoidance. Due to a hardware defect with the CalRadio platform, however, we are not able to faithfully implement carrier

sense. Our implementation is sufficient to exchange packets both between CalRadios and with other, Atheros-based 802.11b devices in our lab, but suffers from an unusually high loss rate due to lack of carrier sense. We have attempted to ameliorate this issue by introducing a fixed, per-station delay after the completion of a previous transmission to avoid frequent collisions. While this slotting mechanism does not interfere with the operation of RTS-id, it has the unfortunate effect of decreasing the effective channel utilization. When RTS(-id)/CTS is enabled, however, this limitation impacts only the RTS/CTS exchange, as the successful completion of such an exchange will reserve the channel for data transmission.

5 Infrastructure networks

We use our testbed to show RTS-id's ability to optimize "node-to-node" transmissions between nodes communicating through the same access point, finding that RTS-id reduces the number of data frame transmissions by 50.7% and improves bulk UDP throughput by 26.1% in our testbed configuration. These results translate into a 25–46% reduction (depending on data rate) in air time usage compared to a network that does not use RTS/CTS.

5.1 Node-to-node transmission

When two nodes on the same infrastructure-based wireless network communicate with each other, they must relay their packets through an AP with which they are associated, even if they are within transmission range of each other. RTS-id provides a natural mechanism to optimize this communication by allowing the AP to short-circuit its retransmission of the packet.

We are not aware of empirical data quantifying the overhearing prevalence in typical access point deployments. Hence, we attempt to emulate realistic situations such as meetings or office collaborations by setting up three nodes in a controlled topology. We physically connect three nodes together through a series of splitters and variable attenuators so that the path loss between *A* and *B* is *L* dB, *B* and *C* is 20 dB, and the loss between *A* and *C* is $(50 + L)$ dB. We have found that our CalRadios can tolerate path loss of approximately 100 dB in our noise-free configuration, so we can control the prevalence of overhearing by adjusting the value of *L*.

Node *A* is configured to use node *B* as its first-hop router. Node *B* plays the role of an access point by forwarding *A*'s packets on to node *C*. We use the `tcp` application to send 1100-byte UDP packets and report our results both in terms of individual frame exchanges and path throughput. To reduce the impact of external nodes, we set the CalRadios to channel nine, a relatively quiet

Node	Tx success	CTS-ACK	CTS	ACK
A	99.3%	0%	56.6%	99.9%
B	98.6%	0%	45.0%	99.9%
110-dB path loss : 2.05 data frames per packet, 29.13 KBps				
A	99.7%	0.1%	96.6%	99.8%
B	99.9%	97.6%	1.1%	100%
100-dB path loss: 1.01 data frames per packet, 36.74 KBps				

Table 2: Experimental results from the CalRadio test-bed.

channel in our building. All three nodes support RTS-id. Node *A* first sends the packet to node *B*. The Linux networking stack on node *B* then forwards the packet to node *C*. Meanwhile, node *C* is promiscuously listening to all packets; since all three nodes are in close physical proximity, *C* frequently overhears *A*'s transmissions to *B*. In such cases, it caches the packet and records the packet ID. When *B* subsequently sends an RTS-id frame to *C* requesting to transmit a packet with an ID that *C* just overheard, *C* delivers the cached copy to the Linux kernel and responds with a CTS-ACK preventing the transmission of the data frame. If *C* did not overhear the original transmission, it sends a CTS, and *B* transmits the data frame to *C*, which acknowledges its receipt and delivers the packet to the application.

5.2 Transmission reduction

To demonstrate the effectiveness of RTS-id, we conduct two separate experiments with drastically different overhearing rates. In the first, we set the variable attenuator to $L = 60$ dB, resulting in a path loss from *A* to *C* of 110 dB, effectively preventing overhearing. In the second, we adjust the attenuator to 50 dB, giving an effective path loss of 100 dB which results in significant overhearing. Both experiments attempt to transmit a train of UDP packets from *A* to *C* at 1 Mbps with RTS-id enabled. We set the link-layer retransmission count to ten, meaning a sender will attempt the RTS-id/CTS/data/ACK frame exchange at most ten times for each packet.

Table 2 presents the results of these experiments. For each node, we show the fraction of attempted packet transmissions successfully completed by that node, as well as the fraction of RTS attempts that were met with either a CTS-ACK (and therefore avoided) or a regular CTS (and therefore transmitted). Finally, we show the percentage of transmitted data frames that were successfully acknowledged by the receiver.

Due to lack of carrier sense, RTS/CTS exchanges fail relatively frequently in our experiment, especially without overhearing. Recall that the frame exchange will be attempted up to ten times for each packet, so the overall transmission success rate is still quite high. In contrast, almost no data frames are dropped. The stark difference

in RTS/CTS success rates between the two experiments is due to the fact that node *B* rarely needs to transmit data frames in the overhearing case, so there is far less contention for the channel.

As expected, node *C* overhears a large fraction of the transmissions from *A* to *B* when $L = 50$ dB; hence, it is able to prevent all but 1.1% of the packets from being forwarded by *B*. Comparing the overhearing case with the non-overhearing case, RTS-id provides dramatic savings, reducing the number of data frames transmitted per successfully delivered packet from just over 2.05 (recall that 2.0 is the best case without overhearing if there is no data frame loss) to 1.01, a 50.7% reduction in transmission rate, which resulted in a 26.1% improvement in end-to-end bandwidth in our testbed configuration.

5.3 RTS/CTS overhead

Most infrastructure deployments do *not* enable RTS/CTS by default; as a result, using our adaptive algorithm an AP will only enable RTS-id if the expected savings outweigh the additional overhead (Section 3.3). Due to the lack of carrier sense, we are unable to effectively measure the performance improvement in this scenario. Using statistics collected from the experiments depicted in Table 2, however, we can calculate the air time usage for a non-RTS-id network from the non-overhearing case by simply summing the amount of air time used by the data transmissions (DIFS + data + SIFS + ACK), as RTS/CTS frames would not be used in this case. Conversely, we can calculate the total air time usage for an adaptive RTS-id deployment by summing the air time used by the data transmissions from *A* to *B* in the overhearing case and combining that with the data transmissions and RTS-id/CTS frames from *B* to *C* (DIFS + RTS-id + SIFS + CTS-ACK) or (DIFS + RTS-id + SIFS + CTS + SIFS + data + SIFS + ACK). Considering the 1100-byte packets transmitted at 1 Mbps in the previous experiment, an RTS-id enabled network would use 46.1% less air time than one not using RTS/CTS at all. The savings reduce to 25.2% if one considers MTU-sized packets at 11 Mbps.

6 Mesh networks

Due to the limited availability of CalRadios, we use trace-based simulation to evaluate the effectiveness of RTS-id in a multi-hop mesh network. Its benefits in this scenario range from a 20% savings for the median route at 1 Mbps to a 12% savings for the median route at 11 Mbps. In general, we find that RTS-id benefits even highly optimized routing mechanisms, but that its benefit is somewhat inversely proportional to how optimal the route choice and—more significantly—rate and power selection is. This

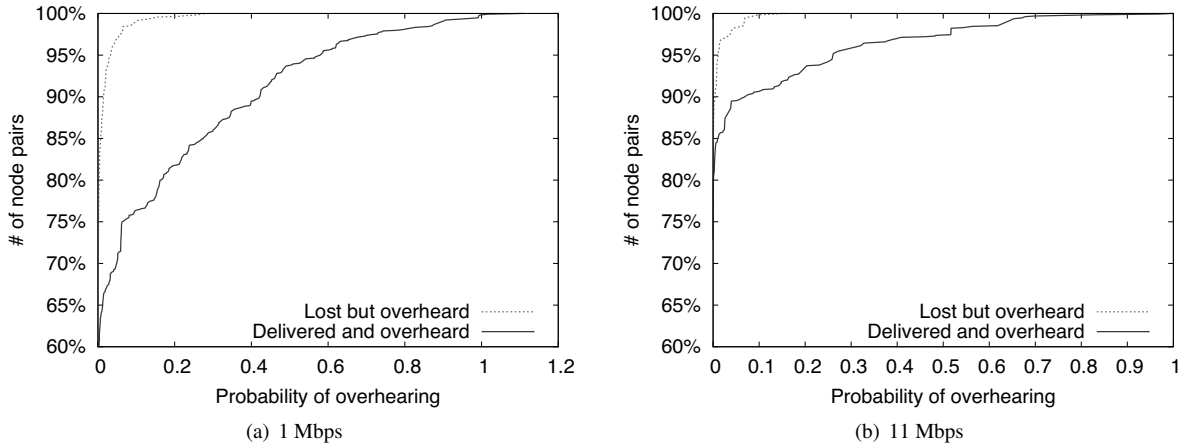


Figure 4: Overhearing in Roofnet. We plot the probability that any transmission along an ETT path is overheard by a node *further* along the path. We plot two mutually exclusive cases: when intended destination does and does not also receive the packet. Both y axes start at 60%.

follows intuitively: a large amount of overhearing along a transmission path is a possible signal that the sender is transmitting “too well” to reach the receiver, and so could perhaps spend that extra signal/noise ratio by using a faster transmission rate or lower power.

Our mesh evaluation first considers how often a node can overhear transmissions in realistic environments at fixed rates, and how that impacts the number of transmissions required to forward a packet using the popular ETT routing metric [1]. We then evaluate the effect of rate adaptation and alternate traffic patterns. Next, we examine how RTS-id provides greater benefit to less sophisticated route selection metrics, and then evaluate the savings provided by RTS-id in an environment that does not use RTS/CTS by default.

Dataset: Our analysis uses the Roofnet mesh network dataset [1]. The dataset contains the fraction of MTU-sized packets transmitted at each node that are received at every other node. In other words, the dataset specifies $\Pr_r[A \rightarrow B] \forall A, B \in G, r \in \{1, 2, 5.5, 11\}$ Mbps. The dataset was collected on the 38-node MIT Roofnet network as follows:

Each node in turn sends 1500-byte 802.11 broadcast packets as fast as it can, while the rest of the nodes passively listen. Each sender sends for 90 seconds at each of the 802.11b bit-rates. The experiment uses 802.11 broadcast packets because they involve no link-level acknowledgments or retransmissions. [1]

The reception rates were measured with only one Roofnet node was transmitting at a time—though there likely were other 802.11 sources during the experiment. It is possible that simultaneous Roofnet transmissions

would decrease the rate of overhearing as the load on the network increases, but it unclear how significant this effect would be. Unfortunately, there are no published Roofnet datasets under such conditions.

Route computation: Unless specified otherwise, we compute routes using a modified ETT metric [4], which roughly approximates the expected transmission time (including acknowledgments, retries, and back-offs) required to successfully transmit a packet across a given link. ETT is derived from the expected transmission count (ETX) [11], which has been shown to outperform previous routing metrics [12]. The ETX metric is defined for each pair of nodes at rate r , and is computed as $1/(p_f \cdot p_r)$, where p_f is the transmission success rate in the forward direction (i.e., $\Pr_r[A \rightarrow B]$), and p_r is the success rate in the reverse direction ($\Pr_r[B \rightarrow A]$). A key distinction between traditional ETX and ETT is that, in ETT, p_r is based upon the measured delivery rate of 60-byte ACK packets transmitted at one Mbps. Unfortunately, the 2004 dataset does not include the 60-byte loss data necessary to calculate ETT; hence, we modify ETT slightly to always consider the delivery rate on the reverse channel at one Mbps, but are forced to use the rate for 1500-byte packets, which is likely to be lower. We then use a shortest-path algorithm to find the path between each pair of nodes that minimizes the total ETT metric.

6.1 Overhearing prevalence

Overhearing is common in the Roofnet topology, particularly at lower speeds. We compute the probability of overhearing by all node pairs that occur together on some valid source-destination route in the topology. To do so, we create a superset distribution of packet reception $\Pr_r[A \rightarrow \{B, C\}], \Pr_r[A \rightarrow \{B, C, D\}] \dots$, the probability

that a packet transmitted by A to B at rate r is received by all possible combinations of receivers $\{B, C\}$, $\{B, C, D\}$, etc.

Figure 4 shows the CDF of the overhearing probabilities for paths between each pair of nodes in the network. We consider all ETT paths $P \in G$ longer than one hop, where $P := X_1 \rightarrow X_2 \rightarrow \dots X_n$, and compute the probability that any transmission along the path is overheard by a node further along the path. That is, X_i 's transmission to X_{i+1} is overheard by X_j , $j > i + 1$. There are two cases of interest: where X_{i+1} does not and does also receive the transmission. Our current implementation of RTS-id does not immediately assist in the first case where the packet is overheard but not delivered to its intended next hop. The packet will need to be retransmitted by the sender until it has been received at the next-hop—although the subsequent transmission by the next-hop will be avoided. While it may be possible to extend RTS-id to prevent these retransmissions, doing so would require knowledge of the intended route, and the situation is unlikely to occur frequently in practice with reasonable route selection. Indeed, it occurs only rarely in the Roofnet dataset. Hence, for simplicity, we forgo the seemingly minimal potential performance improvement and only act upon packets that are both overheard and successfully received by their intended recipient. Transmissions between a fifth of all node pairs are overheard more than 20% of the time at 1 Mbps. Overhearing is less common at higher speeds. At 11 Mbps, only 5% of node pairs are overheard more than 20% of the time. In an outdoor environment like Roofnet, however, nodes frequently transmit at lower link rates, so ample opportunity exists to exploit overhearing.

6.2 Transmission reduction

To evaluate whether RTS-id can exploit overhearing and ACK loss to avoid transmissions, we construct a statistical model to estimate the expected number of transmissions along each path. We examine each source/destination pair individually, and for each pair:

1. Create a state machine in which each state corresponds to the set of nodes that have heard a given packet. For example, if a route has three hops: $A \rightarrow B \rightarrow C \rightarrow D$, the initial state is A and the final state is $ABCD$.
2. Next, calculate the probability for each state transition under normal 802.11 and using RTS-id. Initially, we neglect the RTS/CTS exchange, and consider only data packets and link-layer ACKs. Transitions exist between a node and itself (self-loops due to failed transmissions, regardless of overhearing), adjacent nodes on the path (successful normal transmissions) and, for RTS-id, a node and all subsequent

nodes in the path (due to overhearing). For the base 802.11 case, we consider a transmission successful if the packet reaches the receiver and the corresponding ACK reaches the sender; the probabilities are drawn from the Roofnet data set. For RTS-id, we ignore the ACK delivery rate because any spurious retransmission attempts will be bypassed by RTS-id, and compute state transition probabilities based upon the overhearing distribution. For simplicity, in each state we assume that the packet is only transmitted by the node furthest along the path.

3. Finally, calculate the expected number of transitions (i.e., packet transmissions) required to reach the last state (where the destination has received the packet) from the first state. We compute the expected number of transmissions twice, once using the RTS-id transitions and probabilities, and once using only the on-path $A \rightarrow AB$ and $AB \rightarrow ABC$ base-case 802.11 transitions.

Without overhearing, only two transitions leave each state: $AB \rightarrow ABC$ for successful delivery, and $AB \rightarrow AB$ for failure. With overhearing, the picture is far more interesting. Figure 5 shows four state machines corresponding to actual paths in the Roofnet dataset. Figure 5(a) depicts a path with no overhearing; that is, C never overhears A 's transmission, therefore the only possible transition is from A to AB , which occurs 92.65% of the time (the other 7.35% of the time the packet is lost and must be resent). The link from B to C is much worse, and succeeds less than 60% of the time. Figure 5(b) shows a simple overhearing scenario, where 12.85% percent of the time A 's transmission to B is overheard by C .

The remaining two examples depict more complicated transitions that occur with longer paths. Figure 5(c) shows a case in which roughly 20% of the time, a packet can be transmitted directly from A to D , obviating the need to forward through B or C . The careful reader may wonder why ETT selected B rather than C as the first hop in the path, as $A \rightarrow C$ appears to have the higher success probability. In this case, the return path (not shown) from C to A is quite lossy, so ETT correctly avoids this hop because the ACKs will be lost. RTS-id, on the other hand, benefits from this overhearing because it does not need to ACK directly from C to A . Finally, Figure 5(d) shows three distinct overhearing paths from A to E : $A \rightarrow B \rightarrow E$, $A \rightarrow D \rightarrow E$, and $A \rightarrow C \rightarrow D \rightarrow E$.

Figure 6(a) plots the expected number of transmissions for all-pairs paths of length greater than one. We omit the one-hop paths for clarity, although we note that the savings is non-zero due to avoided spurious retransmissions. Without RTS-id, each path requires at least as many transmissions as there are hops, sometimes many more due to failed transmissions. RTS-id is able to significantly de-

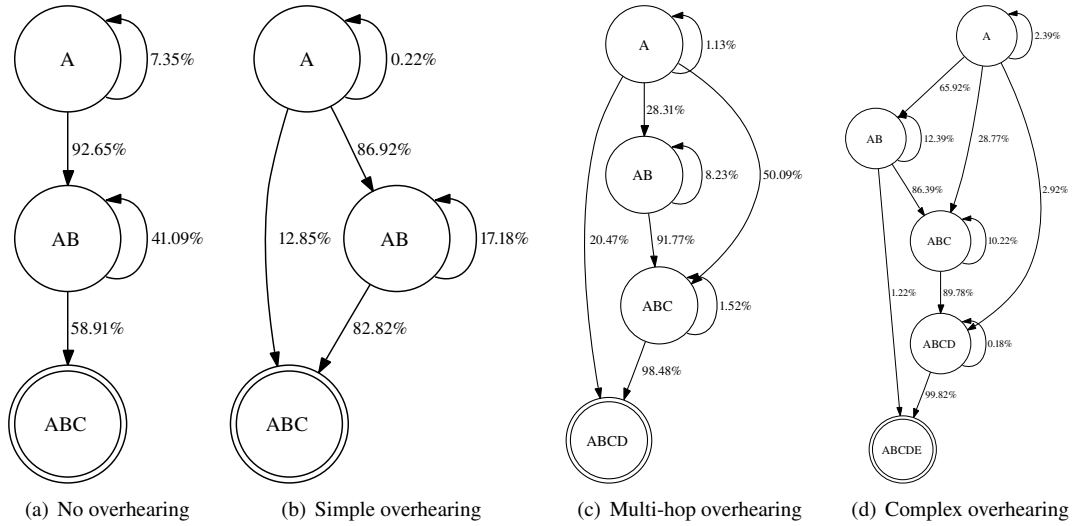


Figure 5: Actual RTS-id scenarios from the Roofnet dataset. Self-loops represent complete packet loss events. All probabilities are based upon a 1-Mbps transmission rate.

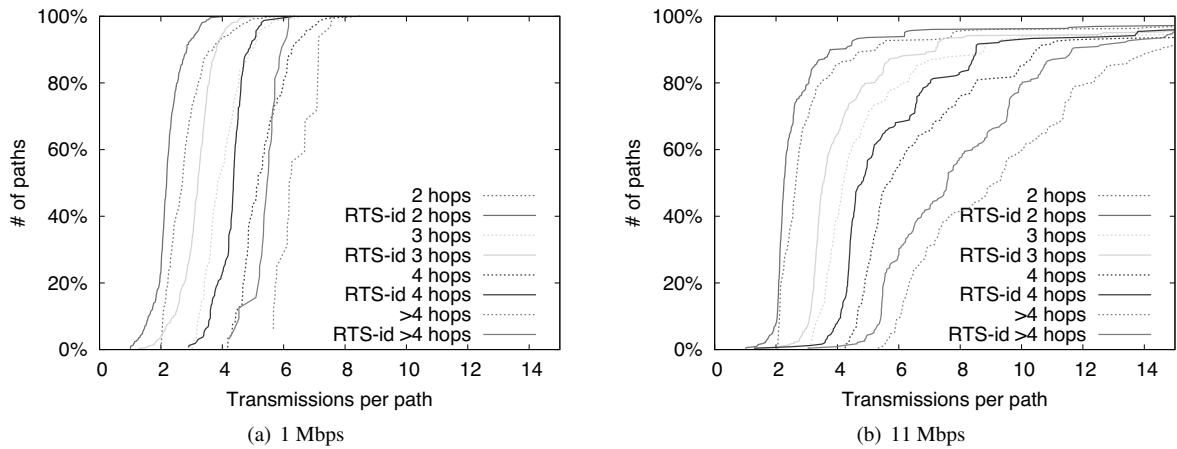


Figure 6: The expected number of packet transmissions per ETT path with and without RTS-id.

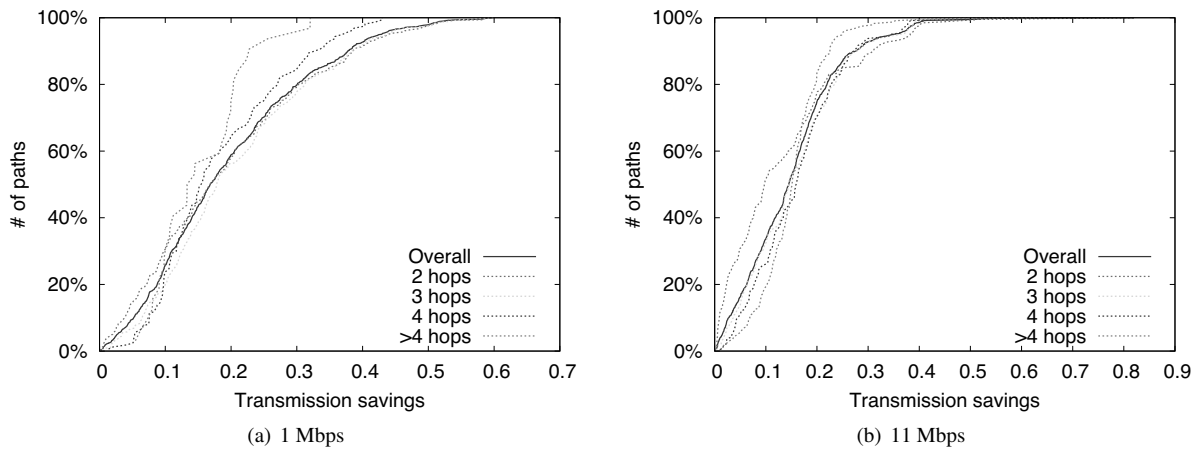


Figure 7: RTS-id performance improvement versus ETT on the Roofnet dataset. The graphs plot the CDF of the fraction of transmissions saved per path for 1 and 11 Mbps transmission rates.

crease the number of transmissions required, often quite dramatically. To clearly illustrate the performance improvement of RTS-id, Figure 7 plots both the relative performance improvement for various path lengths at 1 Mbps and 11 Mbps. At 1 Mbps, RTS-id is able to save over 20% of path transmissions for the median path, and more than 40% (i.e., turn a 5-hop path into a 3-hop path) for over 10% of the paths. Due to the restricted overhearing at 11 Mbps, however, RTS-id provides at least 20% savings for only a quarter of all paths.

6.2.1 Rate adaptation

As the previous results showed, RTS-id is more effective with lower transmission rates that can reach more nodes. Choosing transmission rates on a per-node basis is complex: higher rates have smaller reception distances, and so may require more hops through the ad hoc network. Here, we model Roofnet’s “SampleRate” technique for rate selection [? 4]. For each link, SampleRate selects the bit-rate with the lowest instantaneous ETT metric. While Roofnet can adjust transmission rates on a per-packet basis, it constructs routes using long-term averages. Hence, we compute an ETT-based path for each source/destination pair as before, except that each hop uses the bit-rate selected by SampleRate. The resulting routes approximate those used by the current version of Roofnet except that we again use the 1500-byte 1-Mbps loss rate for the return channel.

Figure 8 plots both the overhearing prevalence (c.f. Figure 4) and the relative performance improvement versus ETT (c.f. Figure 7) with dynamic rate adaptation. It turns out that most links in our dataset select the 11 Mbps transmit rate, so the overhearing is closer to that observed with a constant 11-Mbps transmit rate than a 1-Mbps transmit rate, resulting in similar savings.³ In particular, RTS-id provides more than 20% savings for one quarter of all routes, and over 35% savings for the most-improved 5%.

6.2.2 Actual traffic patterns

So far, we have considered all source/destination pairs, which is reasonable for many mesh networks. Some mesh networks (e.g., Roofnet), however, rarely route traffic between internal nodes; instead, they forward traffic to and from a few gateway nodes that transfer packets to the Internet. To confirm that our results are not biased by poorly-performing internal routes, and, instead, are representative of the paths traversed by actual traffic, we restrict ourselves to only those paths connecting each Roofnet node to each of the four Roofnet gateway nodes.

³Interestingly, its designers note that Roofnet generally transmits at 5.5 Mbps in practice [4], so we are likely understating the potential savings.

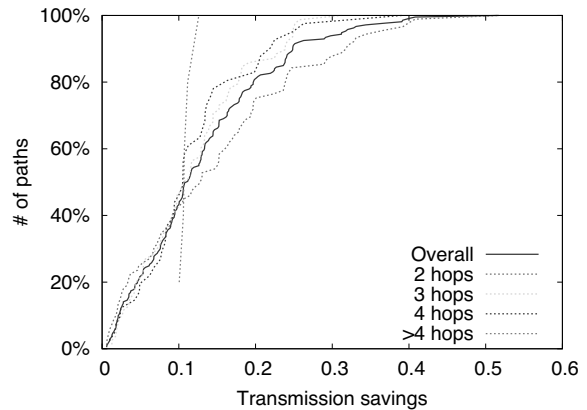


Figure 9: The relative performance improvement versus ETT for paths leading to or from a Roofnet gateway.

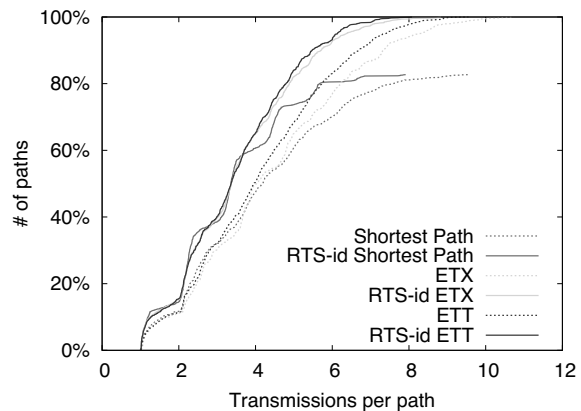


Figure 10: A variety of routing protocols with and without RTS-id, all using SampleRate to select link transmission rates.

Because we do not have a traffic matrix, we consider paths to all four gateways from every node, although only one of them is likely used at any point in time. Figure 9 shows the same data as Figure 8(b), except that it contains only gateway routes. The overall distribution of savings is roughly unchanged.

6.3 Improving other routing protocols

In general, RTS-id improves the performance of routing *more* if those routing protocols do not select routes optimally. Our evaluation of RTS-id using ETT (currently the best-performing routing protocol available for mesh-based networks) gives ETT a large advantage, assuming that ETT has perfect knowledge of link loss rates and that those loss rates are stationary. Our ETT routes are computed as the optimal value over the entire 90-second measurement. In practice, however, networks cannot devote all of their resources to measurement.

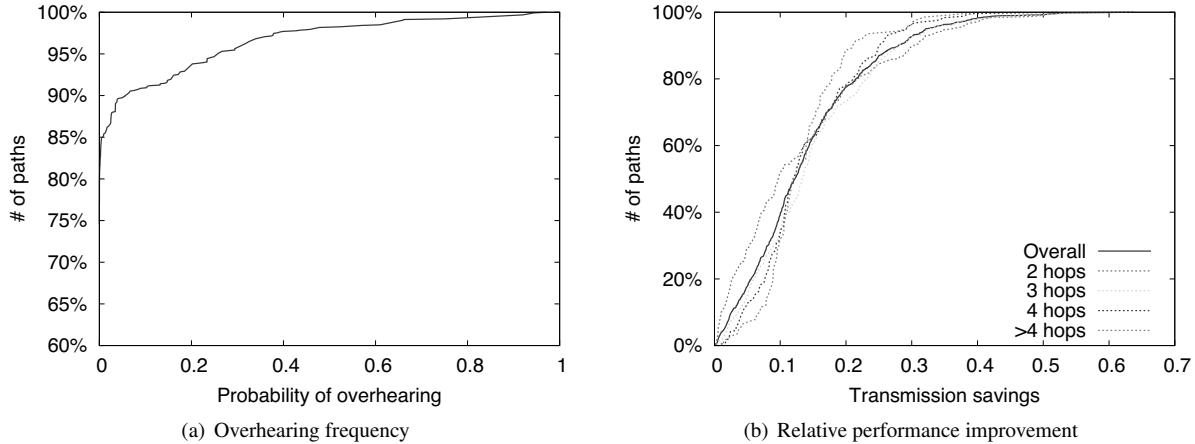


Figure 8: The impact of rate adaptation. The first graph shows the overhearing prevalence (c.f. Figure 4), and the second shows the relative performance improvement versus ETT.

For example, the Roofnet network computes its metrics using only 10 measurement packets sent every five minutes, leading to less accurate information for route construction. Furthermore, many networks currently operate with much simpler protocols that do not need to collect such fine-grained loss information. Here, we demonstrate that not only does RTS-id substantially improve the performance of these routing protocols, but that RTS-id, operating only on a local per-link basis, raises the performance of other routing protocols above and beyond ETT's performance.

Figure 10 shows the performance of three routing protocols, ETT (c.f. Figure 8), ETX, and shortest path, where shortest path simply selects the path between source and destination with fewest hops, assuming the link delivery rate is above 80%. (80% is arbitrary, and results are similar for other cut-offs.) Note that not all node pairs are connected by paths consisting entirely of links with greater than 80% delivery rates, so the shortest path algorithm constructs fewer routes. For each routing protocol, we plot the absolute number of expected transmissions per path with and without RTS-id. Note that any routing protocol with RTS-id is generally superior to the best protocol (ETT) without it.

6.4 RTS/CTS overhead

As noted earlier, RTS/CTS is not commonly used in infrastructure deployments (though in some, CTS-to-Self packets are sent for 802.11b/g compatibility). While it was designed for multi-hop scenarios, some mesh networks also eschew its use [4], particularly those with infrequent contention. As in the single AP study, enabling RTS-id in these scenarios also requires an extra RTS/CTS exchange, so we again quantify the transmission time required for all packets in the transmission.

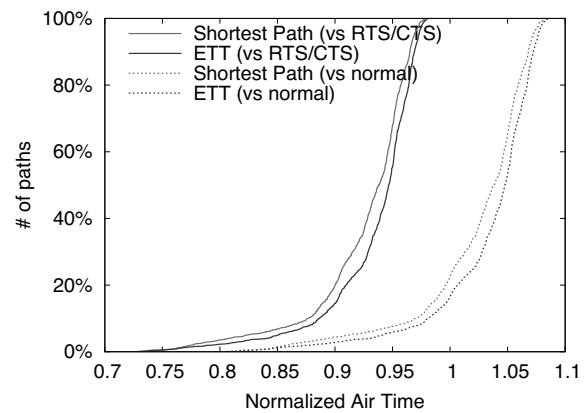


Figure 11: The normalized total path transmission time for RTS-id, with and without RTS/CTS.

We measure this overhead in the Roofnet dataset by examining the path transmission time (the sum of all transmission times along the path). We plot this transmission time normalized against two baselines: a network using no RTS/CTS at all, and a network that already uses RTS/CTS. Note that in this simulation, there is no contending traffic, and so no opportunity for RTS/CTS to provide any benefit. Figure 11 shows the CDF of this normalized transmission time when we do *not* adaptively enable or disable RTS-id and simply leave it enabled on all links. The two lines on the left of the graph show that RTS-id improves transmission times greatly when the network already uses RTS/CTS; the two lines on the right of the graph show the overhead of enabling RTS/CTS and show that in some cases, blindly enabling RTS-id can *reduce* performance over the base network. Some of the paths, however, still benefit from RTS-id, by up to 20%. (The left pair of lines are represent the same data as the ETT and shortest-path lines from Figure 10.)

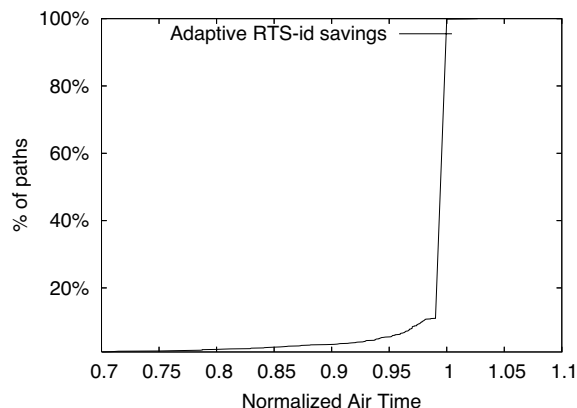


Figure 12: The normalized air time of adaptive RTS-id vs. a network that does not natively use RTS/CTS.

Adaptively enabling RTS-id as described in Section 3.3 avoids the slowdown on links where RTS-id does not provide benefits. To evaluate adaptation, we enable RTS-id only for those link-layer senders who benefit in expectation. Figure 12 shows the fraction of the path transmission time for adaptive RTS-id vs. a network that does not use RTS/CTS at all. The higher overhead of the RTS/CTS exchange means that RTS-id is used on many fewer links than in a network that natively uses RTS/CTS. As a result, its benefits are smaller, but it still provides a 10% reduction in air time for about 5% of the paths, with significantly larger reduction for some paths. Unlike the equivalent lines in Figure 11, adaptive RTS-id never *harms* transmission time.

7 Security implications

The deployment of RTS-id would have a number of potential security implications.

Confidentiality: RTS-id may permit an attacker to perform a rough “traceback” to the source of a packet via timing analysis that determines if a packet was already present in a node’s cache. The effects of such an attack seem minor, as the attacker would already have to be well-placed in order to conduct the inquiry.

Integrity: RTS-id introduces no new attacks against integrity, but the availability attacks discussed below might permit an attacker to prevent legitimate packets from reaching their destination, enhancing the effectiveness of existing spoofing attacks.

Availability: If an attacker knows the hash of the packet it wishes to stop and can generate a packet that collides with this hash, then the attacker can transmit this packet *before* the real packet, causing the attack packet to take the place of the original packet. This attack may require less power than a jamming attack, and the packet

loss would not be detected at link layer, but only end-to-end. A second attack, with similar effect, is that an attacker can spoof a CTS-ACK response to an RTS-id packet, making the sender believe the packet has been transmitted. All of these attacks require that an attacker be able to transmit packets with very tight timing requirements, which today requires programmable hardware such as the CalRadio. While these attacks are somewhat more effective than jamming, an attacker who can mount them is already well-positioned to jam and snoop.

Per-pair keys could help resist these attacks, but their use would require significant modification to the 802.11 protocol: current encryption mechanisms such as WPA only encrypt the data payload, not the packet header.

8 Future work

Our immediate next step is to extend RTS-id to support longer-duration, cross-flow caching. In particular, we would like to integrate Santos and Wetherall-style packet caching with header compression into RTS-id. While existing header compression techniques can compress TCP/IP headers down to a few bytes, they typically rely on tight sender-receiver synchronization; adapting those techniques to the lossy wireless environment poses an interesting challenge. Such extensions could exploit either long-term caching between different flows, or could use small caches to enable efficient simulcast of data over a wireless mesh network without native multicast support.

Our initial evaluation of RTS-id using the Roofnet data leaves several issues unexplored. For instance, how might the performance of RTS-id change in the face of mobility? In particular, the effectiveness of receiver caches may be impacted as the set of overhearing nodes continually changes. Similarly, senders may adjust their transmission power as nodes move, which may increase the need to adaptively enable RTS-id. 802.11 deployments with high levels of mobility, however, may also require higher densities to ensure pervasive connectivity, which could increase overhearing opportunities.

Additionally, our current deployment is restricted to 802.11b. The availability of additional speeds in 802.11g and 802.11a may affect overhearing depending on senders’ rate adjustment algorithms. Moreover, it could be possible to improve the performance of rate adaptation algorithms by integrating information from RTS-id. In particular, it may be beneficial to transmit at a lower rate with significantly higher overhearing; conversely, a sender may not want to decrease its send rate even in the face of significant link-layer loss if overhearing is able to compensate for a large enough portion of the losses. We hope to explore these issues with increasing capability and availability of CalRadio.

RTS-id need not operate independent of other advances that leverage wireless broadcast. Like the batching in ExOR, RTS-id might be able to operate more efficiently if it could batch queries *when multiple packets are in its queue*, without increasing end-to-end latency. We believe that there are also interesting possibilities for merging RTS-id's opportunistic overhearing benefits with the exploitation of omni-directional reception by network coding techniques.

9 Conclusion

RTS-id is a simple, backwards-compatible, link layer mechanism for eliminating redundant packet transmissions in wireless networks. Its goal is to turn broadcast reception from a handicap into an advantage, and our evaluation shows that it succeeds at this goal. RTS-id can improve performance by 5% to 30% in existing networks. Perhaps the most important facets of RTS-id, however, is that it boosts the performance of simpler ad hoc routing schemes so that they match the performance of more sophisticated (and complex) schemes, and that it optimizes a common case of same-LAN data transmission.

In addition to our simulation results using the Roofnet data, we have implemented and conducted a preliminary evaluation of RTS-id on real, interoperable 802.11 hardware. While necessarily limited by the limited availability of this hardware and its relative immaturity, our implementation shows that RTS-id can be practically implemented to meet the timing constraints of 802.11 hardware and can reduce redundant packet transmissions in a real-world setting.

Acknowledgments

The authors wish to thank Jeff Pang and Yu-Chung Cheng for comments on earlier drafts. This work is supported in part by Ericsson Research and Qualcomm through the UCSD Center for Networked Systems (CNS) and the UC Discovery program, and by NSF award CNS-0546551.

References

- [1] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.
- [2] A. Armon and H. Levy. Cache satellite distribution systems: Modeling, analysis, and efficient operation. *IEEE JSAC*, 22(2), Feb. 2004.
- [3] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang. MACAW: A Media-Access Protocol for Packet Radio. In *Proc. ACM SIGCOMM*, London, England, Aug. 1994.
- [4] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Proc. ACM Mobicom*, Cologne, Germany, Sept. 2005.
- [5] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. In *Proc. 2nd ACM Workshop on Hot Topics in Networks (Hotnets-II)*, Cambridge, MA, Nov. 2003.
- [6] S. Biswas and R. Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.
- [7] S. Z. Biswas. Opportunistic routing in multi-hop wireless networks. Master's thesis, Massachusetts Institute of Technology, Mar. 2005.
- [8] California Institute of Telecommunications and Information Technology (CalIT2). CalRadio 1.0. <http://calradio.calit2.net/calradiol.htm>.
- [9] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading structure for randomness in wireless opportunistic routing. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.
- [10] R. R. Choudhury and N. Vaidya. MAC layer anycasting in wireless networks. In *Proc. 2nd ACM Workshop on Hot Topics in Networks (Hotnets-II)*, Cambridge, MA, Nov. 2003.
- [11] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proc. ACM Mobicom*, San Diego, CA, Sept. 2003.
- [12] R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.
- [13] J. Jubin and J. Tarnow. The DARPA Packet Radio Network Protocols. *Proc. of the IEEE*, 75(1):21–32, Jan. 1987.
- [14] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. XORs in the air: practical wireless network coding. In *Proc. ACM SIGCOMM*, pages 243–254, Pisa, Italy, Aug. 2006.
- [15] P. Larsson. Selection diversity forwarding in a multihop packet radio network with fading channel and capture. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(4):47–54, 2001.
- [16] J. Li, C. Blake, D. S. J. De Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *Proc. ACM Mobicom*, pages 61–69, Rome, Italy, July 2001.
- [17] B. Raman and K. Chebrolu. Revisiting MAC design for an 802.11-based mesh network. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, San Diego, CA, Nov. 2004.
- [18] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proc. USENIX Annual Technical Conference*, New Orleans, LA, June 1998.
- [19] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking*, 10(6), Dec. 2002.
- [20] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, Sept. 2000.

Beyond Pilots: Keeping Rural Wireless Networks Alive

Sonesh Surana*

Rabin Patra*

Sergiu Nedeveschi*

Manuel Ramos†

Lakshminarayanan Subramanian‡

Yahel Ben-David§

Eric Brewer* ¶

Abstract

Very few computer systems that have been deployed in rural developing regions manage to stay operationally sustainable over the long term; most systems do not go beyond the pilot phase. The reasons for this failure vary: components fail often due to poor power quality, fault diagnosis is hard to achieve in the absence of local expertise and reliable connectivity for remote experts, and fault prediction is non-existent. Any solution addressing these issues must be extremely low-cost for rural viability.

We take a broad systemic view of the problem, document the operational challenges in detail, and present low-cost and sustainable solutions for several aspects of the system including monitoring, power, backchannels, recovery mechanisms, and software. Our work in the last three years has led to the deployment and scaling of two rural wireless networks: (1) the Aravind telemedicine network in southern India supports video-conferencing for 3000 rural patients per month, and is targeting 500,000 patient examinations per year, and (2) the AirJaldi network in northern India provides Internet access and VoIP services to 10,000 rural users.

1 Introduction

The penetration of computer systems in the rural developing world has been abysmally low. Several efforts around the world that have tried to deploy low-cost computers, kiosks and other types of systems have struggled to remain viable, and almost none are able to remain operational over the long haul. The reasons for these failures vary, but at the core is an under-appreciation of the many obstacles that limit the transition from a successful pilot to a truly sustainable system. In addition to financial obstacles, these include problems with power and equipment, environmental issues (e.g. heat, dust, lightning), and an ongoing need for trained local staff, as trained staff move on to better jobs.

Researchers (ourselves included) tend to focus on the sexy parts of a deployment, such as higher performance or a highly visible pilot. However, real impact requires a sustained presence, and thus operational challenges must be viewed as a first-class research topic. Analogous to research on high availability, we must understand the actual causes of operational problems and take a broad systemic view to address these problems well.

In this paper, we describe our experiences over the last three years in deploying and maintaining two rural wireless systems based on point-to-point WiFi links. Our prior work on *WiFi-based Long Distance Networks (WiLDNet)* [26] developed a low-cost high-bandwidth long-distance solution, and it has since been deployed successfully in several developing regions. We present real-world validation of the links, but the primary contribution here is the exploration of the operational challenges of two rural networks: a telemedicine network at the Aravind Eye Hospital [3] in southern India and the AirJaldi [1] community network in northern India.

We have had to overcome major challenges in both networks: (1) components fail easily due to low quality power, (2) fault diagnosis is hard because of non-expert local staff and limited connectivity for remote experts, and (3) remoteness of node locations makes frequent maintenance difficult; thus fault anticipation becomes critical. All of these problems can be fixed by having higher operating budgets that can afford highly trained staff, stable power sources, and robust high-end equipment. But the real challenge is to find solutions that are sustainable and low-cost at all levels of the system. To this end, our main contributions are (1) documenting and categorizing the underlying causes of failure for the benefit of researchers undertaking rural deployments in the future, and (2) developing low-cost solutions for these failures.

In overcoming these challenges we have learned three important lessons that we argue apply to IT development projects more broadly. First, designers must build systems that reduce the need for highly trained staff. Second, simple redesign of standard components can go a long way in enabling maintenance at lower costs. And third, the real cost of power is not the grid cost, but is the cost of overcoming poor power quality problems. By applying these lessons to several aspects of our system including

*University of California, Berkeley

†University of the Philippines

‡New York University

§AirJaldi, Dharamsala, India

¶Intel Research, Berkeley

monitoring, power, backchannels, recovery mechanisms, and deployed software, we have made real progress in keeping these rural networks alive.

The Aravind network now uses WiLDNet to interconnect rural vision centers with their main hospitals for patient-doctor video-conferencing. Currently 9 vision centers cater to 3000 patients per month. Thus far, 30,000 rural patients have been examined and 3000 have had significant vision improvement. As all vision centers are now running with no operational assistance from our team, the hospital considers this network sustainable and is targeting a total of 50 centers in the next 2 years. Similarly, AirJaldi is also financially sustainable and currently provides Internet access and VoIP services to over 10,000 users in rural mountainous terrain.

In the next section we validate the sufficiency of real-world WiLD performance, and outline the challenges to operational sustainability. Section 3 provides some background for the Aravind and AirJaldi networks. In Section 4, we document many of our experiences with system failures, and then in Section 5 present the design of all levels of our system that address these issues. Related work is discussed in Section 6, and in Section 7 we summarize three important lessons for rural deployments.

2 Motivation

In this section, we confirm high-throughput performance of WiLDNet links in real-world deployments, and then outline the operational challenges that remain obstacles to sustained impact.

2.1 Real-World Link Performance

Existing work [16, 26, 29, 33, 34] on rural networking has focused on making WiFi-based long-distance point-to-point links feasible. The primary goal has been high performance, typically expressed as high throughput and low packet loss. In prior work, we have studied channel-induced and protocol-induced losses in long-distance settings [33], and have addressed these problems by creating WiLDNet: a TDMA-based MAC with adaptive loss-recovery mechanisms [26]. We have shown a 2–5 fold increase in TCP/UDP throughput (along with significantly reduced loss rates) in comparison to the best throughput achievable by the standard 802.11 MAC. We had shown these improvements on real medium-distance links and emulated long-distance links.

In this paper we confirm the emulated results with data from several real long-distance links in developing regions. Working with Ermanno Pietrosemoli of Fundación Escuela Latinoamericana de Redes (EsLaRed), we were able to achieve a total of 6 Mbps bidirectional TCP throughput (3 Mbps each way simultaneously) over a single-hop 382 km WiLDNet link between Pico Aguila

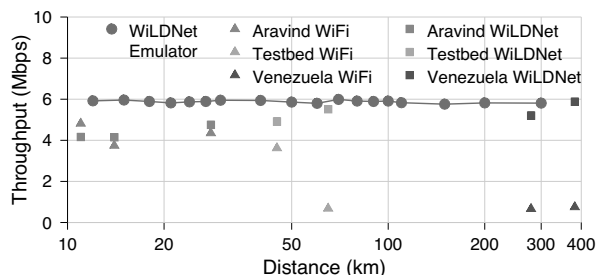


Figure 1: Comparison of TCP throughput for WiLDNet (squares) and standard WiFi MAC (triangles) from links in Aravind, Venezuela, Ghana (the 65 km link), and our local testbed in the Bay Area. Most urban links in Aravind had up to 5–10% loss, and so WiLDNet did not show substantial improvement over standard WiFi. However, WiLDNet’s advantage increases with distance. Each measurement is for a TCP flow of 60s, 802.11b PHY, 11Mbps.

and Platillon in Venezuela. To the best of our knowledge, this is currently the longest distance at which a stable high-throughput WiFi link has been achieved without active amplification or custom antenna design. Each site used a 2.4 GHz 30-dBi reflector grid antenna with 5.3° beam-width and a 400 mW Ubiquiti SR2 radio card with the Atheros AR5213 chipset.

Figure 1 presents results from running WiLDNet on real links from our various deployments in Aravind (India), Venezuela, Ghana, and our local testbed in the Bay Area. We match the performance of WiLDNet over emulated links and greatly exceed the performance of the standard WiFi MAC protocol at long distances.

Thus we find that we are no longer limited by performance over long distances in rural networks. Instead, based on our experiences in deploying and maintaining networks in the two rural regions of India for the last three years, we argue that operational challenges are now the primary obstacle to successful deployments.

2.2 Challenges in Rural areas

Addressing these challenges requires looking at all levels of the system, starting from the power supply and base hardware, up through the software and user interface, all the way to training and remote management. Although remote management, reliable power and training of staff is hard in general, these problems are exacerbated in rural areas for several specific reasons [35].

First, local staff tend to start with limited knowledge about wireless networking and IT systems. This limits their diagnostic capabilities and results in inadvertent misuse and misconfiguration of equipment. Thus management tools need to help with diagnosis and must be educational in nature. The effectiveness of training is limited by the high turnover of IT staff, so education

must be an ongoing process.

Second, the chances of hardware failures are higher because of poor power quality and harsh environments (e.g. exposure to lightning, heat, humidity, or dust). Although we do not have conclusive data about the failure rate of equipment for power reasons in rural areas, we have lost far more routers and adapters for power reasons in rural India than we have lost in our Bay Area testbed. This calls for a solution that provides stable and high quality power to equipment in the field.

Third, many locations with wireless nodes, especially relays, are quite remote, and therefore it is important to avoid unnecessary visits to remote locations. We need to enable preventive maintenance during scheduled visits. For example, evidence of a gradual degradation in signal strength at a remote router could indicate that a cable needs to be replaced or antennas need to be realigned in the course of a normal visit.

Fourth, the wireless deployment may often not be accessible remotely or through the Internet. The failure of a single link might make parts of the network unreachable, even if the nodes themselves are functional. This makes it very hard for remote experts or even local administrators to resolve or even diagnose the problem.

3 Background

Over the last three years we have deployed two rural wireless networks in India. One is at the Aravind Eye Hospital in south India where we link doctors at the centrally located Theni hospital to village clinics, known as vision centers, via point-to-point WiLD links. Patients video-conference over the links with the doctors for consultations. The other is in Dharamsala in north India and is called the AirJaldi network. This network is primarily a mesh with a few long distance directional links that provides VoIP and Internet access to local organizations. Both networks have faced largely similar operational challenges, but with some important differences.

3.1 The Aravind Network

The Aravind network at Theni consists of five vision centers connected to the main hospital in Theni (Figure 2). The network has total of 11 wireless routers (6 endpoints, 5 relay nodes) and uses 9 point-to-point links. The links range from just 1 km (Theni - Vijerani) to 15 km (Vijerani - Andipatti). Six of the wireless nodes are installed on towers, heights of which range from 24–42 m; the others use short poles on rooftops or existing tall structures, such as the chimney of a power plant on the premises of a textile factory. Recently, Aravind has expanded this model to their hospitals in Madurai and Tirunelveli where they have added two vision centers. The network is currently financially viable and a

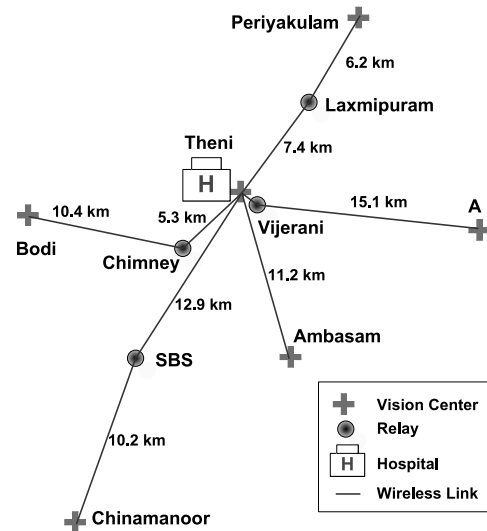


Figure 2: Aravind Telemedicine Network. Theni hospital is connected to 5 vision centers. The other nodes are all relays.

further expansion to 50 clinics around 5 hospitals is being planned to provide 500,000 annual eye examinations.

Hardware: The wireless nodes are 266 MHz x86 single board computers. These routers have up to 3 Atheros 802.11 a/b/g radio cards (200–400 mW). The longer links use 24dBi directional antennas. The routers consume about 4.5W when idle and only 9.5W when transmitting at full bandwidth from 2 radios; 7W is the average power consumption for a node. They run a stripped-down version of Linux 2.4.26 stored on a 512 MB CF card, and include our software for WiLDNet, monitoring, logging, and remote management.

The routers are placed in small and lightweight waterproof enclosures, and are mounted externally, close to the antennas, to minimize signal losses. They are powered via power-over-ethernet (PoE); a single ethernet cable from the ground to the router is sufficient. We use uninterruptible power supplies (UPS) to provide clean power, although we discuss solar power in Section 5.2.

Applications: The primary application is video-conferencing. We currently use software from Marratech [22]. Although most sessions are between doctors and patients, we also use the video conferencing for remote training of staff at vision centers. Typical throughput on the links ranges between 5–7 Mbps with channel loss less than 2%. But 256 Kbps in each direction is sufficient for very good quality video conferencing. Our network is thus over provisioned, and we also use the network to transmit 4–5 MB-sized retinal images. The hospital has a VSAT link to the Internet, but most applications require only intranet access within the network (except for remote management).

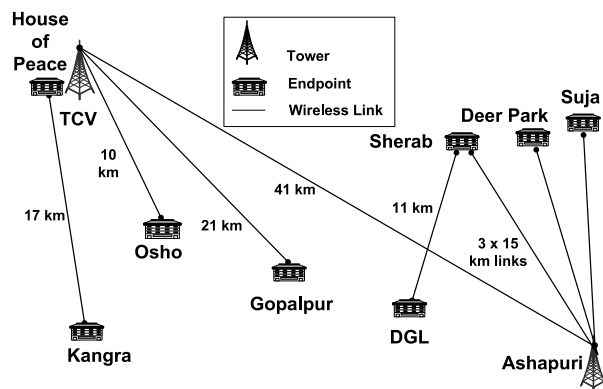


Figure 3: AirJaldi Network. There are 8 long distance links with directional antennas with 10 endpoints.

3.2 The AirJaldi Network

The AirJaldi network provides Internet access and VoIP telephony services to about 10,000 users within a radius of 70 km in rural mountainous terrain characterized by extreme weather. The network has 8 long distance directional links ranging from 10 km to 41 km with 10 endpoints (Figure 3). In addition, the network also has over a hundred low-cost modified consumer access points that use a wide variety of outdoor antennas. Three of the nodes are solar-powered relay stations at remote elevated places with climbable towers. All other antennas are installed on low-cost masts less than 5 m in height; the masts are typically water pipes on the rooftops of subscribers.

Hardware: Most of the routers are modified consumer devices, either Linksys WRT54GL or units from Buffalo Technologies, and cost less than US\$50. They are housed inside locally designed and built weatherproof enclosures, and are mounted externally to minimize signal losses. The antennas, power supplies and batteries are all manufactured locally in India. The router boards are built around a 200MHz MIPS processor with 16 MB of RAM, 4 MB of on-board flash memory, and a low power Broadcom 802.11b/g radio. We run OpenWRT on these routers, and use open source software for mesh routing, encryption, authentication, QoS, remote management and logging. For long distance links and remote relay stations we use slightly higher-end devices such as the PCEngines WRAP boards, MikroTik routerboards, and Ubiquiti LS2s, all with Atheros-based radios.

Applications: The Internet uplink of AirJaldi consists of 5 ADSL lines ranging from 144 Kbps to 2 Mbps for a total of about 7 Mbps downlink and 1 Mbps uplink bandwidth. The longest link from TCV to Ashapuri (41 km) achieves a throughput of about 4–5 Mbps at 2–5% packet loss, while the link from TCV to Gopalpur (21 km) only gets about 500–700 Kbps at 10–15% loss due to the absence of clear line of sight.

This bandwidth is sufficient for applications such as Internet access and VoIP that cater primarily to the needs of the Tibetan community-in-exile surrounding Dharamsala, namely schools, hospitals, monasteries and other non-profit organizations. AirJaldi only provides connectivity to fixed installations and does not offer wireless access to roaming users or mobile devices. A cost-sharing model is used among all network subscribers to recover the operational costs. The network is currently financially sustainable and is growing rapidly.

4 Operational Experiences

We have experienced several operational challenges in both networks that have led to significant downtimes, increased maintenance costs, and lower performance (e.g., increased packet loss). Initially we were involved in all aspects of network planning, configuration, deployment, and maintenance of the networks. Our specific end goal has been to ultimately transfer responsibility to our rural partners, primarily to ensure local buy-in and long-term operational sustainability. This process has not been easy. Our initial approach was to monitor these networks over the Internet and to provide some support for local management, sometimes administering the network directly (bypassing the local staff whenever required). But enabling remote management has been more challenging than expected because of severe connectivity problems (Section 5.3).

This aspect, combined with the desire to enable local operational sustainability, has led us to design the system with more emphasis on support for local management, a particularly challenging problem given limited local experience. One way in which we have ensured that education remains an ongoing process is by creating a three-tier management hierarchy, in which local IT vendors (called *integrators*) with some expertise in networking were hired to form a mid-level of support between local staff and ourselves. With this tiered approach, the rural staff has gradually learned to handle many issues; the IT vendors still handle some, most notably installation, while our role has reduced from operational responsibility to just shipping equipment. In the last year we have not installed any links ourselves even though both networks have grown. We review this transition in our conclusion.

Although we were prepared to expect problems such as poor connectivity, power outages, and misunderstandings around proper usage equipment usage, the actual extent of these problems has been very surprising, requiring a significant custom design of the system at all levels to address these issues effectively. As a result, the reduced downtimes and lower maintenance costs have resulted in both networks being sustainable enough to pay for their

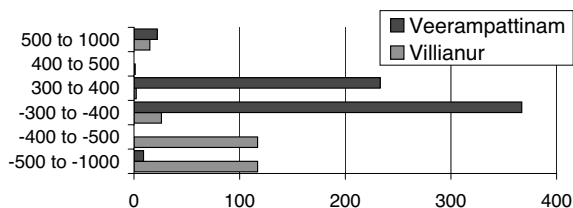


Figure 4: Histogram of power spikes from two rural villages. The bins (y axis) are the size range of the spike in volts, while the x axis is the count. Negative bins imply reversed polarity.

own equipment and towers. Before moving on to the design of our system, we first document three major factors for operational outages; each factor is a result of a combination of the challenges presented in Section 2.2.

4.1 Components Are More Inclined to Fail

Operating conditions at Aravind and AirJaldi have greatly contributed to a substantial decrease in the robustness of system components that would otherwise work quite reliably. One major culprit has been the lack of stable and quality power. Although issues such as frequent power outages in rural areas are well known, we were surprised by the *degree of power quality* problems in rural villages even when power is available. Before addressing the power issues (Section 5.2), not a single day went by without failures related to low power quality in either network. Any effort that is focused on rural deployments must necessarily fix the power issues. Therefore we describe the quality of rural power in detail, particularly because it has not been previously documented.

Low Power Quality: Figure 4 shows data on spikes from a power logger placed in two different rural villages in southern India for 6 weeks. We group the spikes based on their magnitude in volts; negative voltage means the polarity was reversed. We see many spikes above 500V, often with reversed polarity, and some even reaching 1000V! Clearly such spikes can damage equipment (burned power supplies), and has affected us greatly. We have also seen extended sags below 70V and swells above 350V (normal voltage in India is 220-240V). Although the off-the-shelf power supplies we use function well at a wide range of input voltages (80V-240V), they are not immune to such widely ranging fluctuations. Also, locations far away from transformers are subject to more frequent and extreme power fluctuations. Our first approach was to use UPS and battery backups. However, affordable UPS systems are only of the “standby” type where they let grid power flow through untouched; this passes the spikes and surges through to the equipment except during grid outages when the battery starts discharging and is expected to provide stable power.

Failures from Bad Power Quality: We have experienced a wide range of failures from bad power. First, spikes and surges have damaged our power supplies and router boards. In the AirJaldi network, we have lost at least 50 power supplies, about 30 ethernet ports and 5 boards to power surges, while in the Aravind network, we have lost 4 boards, at least 5 power supplies and some ethernet ports as well.

Second, voltage sags have caused brown outs. Low voltages leave routers in a wedged state, unable to boot completely. The on-board hardware watchdog, whose job is to reboot the router, is also often rendered useless because of the low voltages, thus leaving the router in a hung state indefinitely. Third, fluctuating voltages cause frequent reboots, which corrupt and occasionally damage the CF cards through writes during the reboots.

As a typical example, the router at SBS in Aravind rebooted at least 1700 times in a period of 12 months (Figure 5), roughly 5 times per day, going up to 10 times for some days. In contrast, another router at Aravind deployed on top of chimney of a power plant from where it derives reasonably stable power has shown uptimes for several months at a stretch. In practice, we have observed that routers with more frequent reboots are more likely to get their flash memory corrupted over time. We had at least 3 such cases at nodes co-located with the vision centers (Figure 5), which experienced more reboots since staff at these locations shut down and boot up the routers everyday. Finally, frequently fluctuating voltage also prevents optimal charging of the battery backup and halves its overall lifetime.

Lack of quality power increases not only downtime but also maintenance costs. Traveling to remote relay locations just to reboot the node or replace the flash memory is expensive and sometimes has taken us several days, especially in Dharamsala where the terrain is rough.

Other Power-related Problems: In Dharamsala, one of the stormiest locations in India, lightning strikes have often damaged our radios. We have learned the hard way that whenever we deployed a mix of omni and directional antennas, the radios connected to the omni antennas were much more likely to get damaged during lightning storms compared to the radios connected to directional antennas.

It turned out that omni-directional antennas attract lightning more as they are usually mounted on top of masts and have a sharper tip, while directional antennas are typically mounted below the maximum height of the mast. To mitigate this problem, we install omni antennas about 50 cm below the top of the mast. However, this creates dead zones behind the mast where the signal from the antenna is blocked. To reduce these dead zones, we sometimes use an arm to extend the omni antenna away from the mast. After lowering the omni antennas, we have not lost any radios during storms.

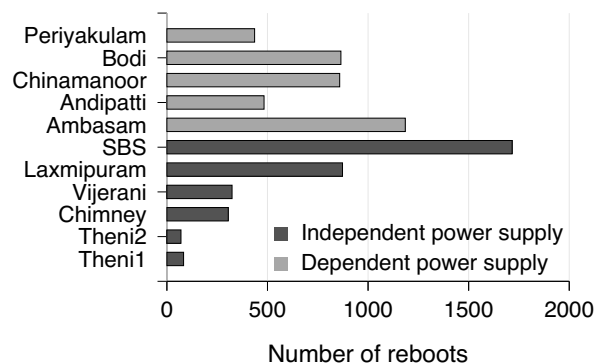


Figure 5: Number of reboots estimated per node in the Aravind network for about one year of operation. Nodes with power supplies dependent on the vision center are turned on or off everyday. Nodes with independent power supplies are typically relay nodes or hospital nodes.

4.2 Fault Diagnosis is Difficult

Accurate diagnosis of the problem can greatly reduce response time and thus downtime. The most common description of a fault by our rural partners is that the “link is down.” There are a wide variety of reasons for network outages and it is not always easy to diagnose the root cause. The lack of appropriate tools for inexperienced staff, combined with unreliable connectivity which hinders detailed monitoring, prevents accurate diagnosis.

For example, a remote host might be running properly, yet is unreachable when an intermediate wireless link goes down. The non-functional link makes it impossible to query the remote host for diagnosis. In fact, there have been many instances where rural staff have traveled to the remote site with great difficulty only to realize that it was a regular power shutdown from the grid (in which case nothing could be done anyway), or that it was a software problem which could have been fixed if there were an alternate backchannel to the router. Accurate diagnosis of such problems can save considerable time and effort, and prevent unnecessary travel. Furthermore, our own ability to help the local staff by logging in remotely to diagnose the problem is limited by connectivity. For instance, we use the VSAT link at Theni (in the Aravind network) to aid the local staff in monitoring and managing the network, but the VSAT backchannel has worked for only 65% of the time in the last one year.

Sometimes local misunderstandings of equipment usage make it even harder to diagnose problems. For example, as shown in Figure 6, an elevator shaft was constructed right in front of the directional antenna at Aravind Theni hospital, completely obstructing the line of sight to the remote end. Whenever we remotely logged in to the Theni end of the link from Berkeley, every-

thing seemed fine except that we could not communicate with the remote end. We had no other network access to the remote host so local staff kept physically checking the remote end, but did not (ourselves included) think of checking the roof at Theni. The resulting downtime lasted for two months until we flew there and saw the problem!

Packet Loss due to Interference: In the AirJaldi network, a decrease in VoIP performance was reported for a particular link at very regular intervals. However without any additional information to diagnose the problem, no action could be taken and this behavior persisted for three months. Finally, after some detailed monitoring by us (and not the rural staff), we saw a regular pattern of packet loss between 8am to 9am every day except Sundays. But scanning the channels showed no external WiFi interference. We were finally able to attribute the problem to a poorly installed water pump that was acting like a powerful spark generator, interfering with wireless signals in the vicinity. Without packet loss information, both the rural staff and we would have had a lot of trouble solving this problem.

Signal Strength Decrease: In the Theni-Ambasam link in the Aravind network (Figure 2), we noticed a drop in signal strength of about 10 dB that persisted for about a month. Without further information it was hard to tell whether the antennas were misaligned, or the pigtail connectors were damaged, or the radio cards were no longer working well. In the end, several different attempts were made by local staff over multiple trips; the radio cards, the connectors and even the antennas were replaced, and the signal strength bumped back up without it being fully clear what finally helped!

Network Partition: We experienced network partitions many times, but for several different reasons. For example, at Aravind, staff misconfigured the routing and added static routes while dynamic routing was already enabled. This created a routing loop partitioning the network. In another instance of operator error, the default gateway of one of the routers was wrongly configured. There were also a few instances when operators changed the IP addresses of the endpoints of a link incorrectly, such that the link was non-functional even though it showed up as being associated. And as mentioned earlier, the construction of the elevator shaft left the network partitioned for two months.

“Fixing” by users: A recurring problem is that well-meaning rural staff often attempt to fix problems locally when the actual root cause is not local. For example, at AirJaldi we have seen that when an upstream ISP goes down, rural staff tend to change local settings in the hope of fixing the problem. These attempts typically create new problems, such as misconfiguration, and in a few



Figure 6: The Theni to Vijerani link in the Aravind network was completely obstructed by a newly constructed elevator shaft. This problem was not resolved until we visited Theni after 2 months.

cases have even resulted in damage to equipment. In all these cases, the network remained non-functional (but now for a different reason) even after the ISP resumed normal connectivity. Thus we need mechanisms to indicate when a link is having problems at the remote end, so as to prevent local attempts at repair.

The general theme is that no matter what the fault, if the link appears to be down with no additional information or connectivity into the wireless node, it is hard for even experienced administrators to resolve the problem.

4.3 Anticipating Faults is Hard

Some of the node locations in our networks, especially relays, are quite remote. Site maintenance visits are expensive, time consuming, and require careful planning around the availability of staff, tools, and other spare equipment. Therefore, visits are generally scheduled well in advance, typically once every six months. In this scenario, it is especially important to be able to anticipate failures so that they can be addressed during the scheduled visits, or if a catastrophic failure is expected, then a convincing case can be made for an unscheduled visit for timely action. But without an appropriate monitoring and reporting system that includes backchannels, it is difficult to prepare for impending faults.

Battery Uptime: At both Aravind and AirJaldi we use battery backups. Loss of grid power at the nodes causes their batteries to start discharging. It is generally not known when the batteries will finally run out. If this information is somehow provided to the staff, they can prevent downtime of the link by taking corrective measures such as replacement of the battery in time. Such feedback would also suggest if the problem were regional (as other routers would also suffer loss of grid power) or site-specific such as a circuit breaker trip.

Problem description	System Aspects
Component Failures	
Unreliable power supply	P
Bad power causing burnt boards and PoEs	P
CF card corruption: disk full errors	M, P, S
Omni antennas damaged by lightning	P
Fault Diagnosis	
Packet loss from interference	M
Decrease in signal strength	M
Network partitions	M, B
Self fixing by users	S
Routing misconfiguration by users	M, B, S
Failed remote upgrade	B, R
Remote reboot after router crash	B, R, S
Spyware, viruses eating bandwidth	M, S
Anticipating Faults is hard	
Finding battery uptime/status	M, B, P
Predict CF disk replacement	M

Table 1: List of some types of faults that we seen in both Aravind and AirJaldi. For each fault, we indicate which aspects of the system, as we have designed it, help mitigate the fault. The different aspects are Monitoring (M), Power (P), Backchannel (B), Independent Recovery Mechanisms (R) and Software (S). The information on faults has been collated from logs and incident reports maintained by the local administrators and remote experts respectively.

Predicting Battery Lifetime: Battery life is limited by the number of deep cycle operations that are permitted. This lifetime degrades sharply because of fluctuating voltages seen in our deployments that do not charge the battery optimally. At Aravind, batteries rated with a lifetime of two years last for roughly three to six months. Information about remaining battery life can also enable prevention of catastrophic failures.

Predicting Disk Failure: We have observed that with frequent reboots over time, the disk partition used to store system logs accumulates bad *ext2* blocks. Unless we run *fsck* periodically to recover the bad blocks, the partition becomes completely unusable very soon. We have also seen that many flash disks show hardware errors, and it is important to keep track of disk errors and replace them before they cause routers to completely fail.

5 System Architecture Design

In this section, we present five aspects of our system: monitoring, power, backchannels, independent recovery mechanisms, and software. Each has been designed to specifically address our goals of increasing component robustness, enabling fault diagnosis, and supporting fault prediction. For each aspect, wherever appropriate, we also discuss tradeoffs affecting our design choices. Table 1 indicates which aspects of our system design are

important for reducing the impact of some of the common faults presented in the previous section.

5.1 Monitoring

All aspects of system management require some level of monitoring. During the initial deployment at Aravind, we faced two main challenges in designing a monitoring system. First, the Aravind network at Theni only allowed us to initiate connections from within the network. Second, local staff was not familiar with Linux or with configuration of standard monitoring software such as Nagios [10].

This led us to build a *push-based* monitoring mechanism that we call “PhoneHome” in which each wireless router pushes status updates upstream to our US-based server. We chose this method over the general *pull-based* architecture in which a daemon running on a local server polls all the routers. The pull-based approach would require constant maintenance via re-configuration of a local server every time a new router would be added to the network. In contrast, the push-based approach enabled us to configure the routers only once, at installation, by specifying the HTTP proxy to be used.

The Aravind network features two remote connectivity options, both of which are slow and unreliable (Section 5.3): (1) a direct CDMA network connection on a laptop at the central hospital node, and (2) a VSAT connection to another hospital, which has a DSL connection to the Internet. PhoneHome is installed on each of the wireless routers. All the routers periodically post various parameters to our US server website. Server-side daemons analyze this data and plot visual trends.

We collect node and link-level information and end-to-end measurements. The comprehensive list of the measured parameters is presented in Table 2. Most of these parameters can be measured passively, without interfering with normal network operation. However, several of these measurements, such as maximum link or path throughput, require active testing. Some of these tests can be performed periodically (e.g. pinging every network host), and some of them are done on demand (e.g. finding the throughput achievable on a particular link at a given time).

We also use the PhoneHome mechanism for remote management. Every time PhoneHome connects to our US server, it opens a reverse SSH tunnel back into the wireless node, enabling interactive SSH access to the Aravind machines. As the VSAT connection only allows access over an HTTP proxy, we are required to run SSH on top of HTTP, and configure PhoneHome with the proxy. In case of a direct connection to the Internet, no such configuration is required. Another option (employed in the remote management of AirJaldi) is to use the OpenVPN software to open VPN tunnels between network routers

Scope	Type	Measured Parameter
Node	Passive	CPU, disk and memory utilization, interrupts, voltage, temperature, reboot logs (number & cause), kernel messages, solar controller periodic data
	Active	disk sanity check
Link	Passive	<i>traffic</i> :, traffic volume(#bytes, packets) <i>wireless</i> : signal strength, noise level, # control packets, # retransmissions, # dropped packets <i>interference</i> : # of stations overheard & packet count from each, # corrupted packets
	Active	liveness, packet loss, maximum link bandwidth
System	Passive	route changes, pairwise traffic volume & type
	Active	pairwise end-to-end delay & max throughput

Table 2: Parameters collected by PhoneHome.

and remote servers.

PhoneHome proved to be helpful in understanding failures, diagnosing and predicting many faults. First, it helped maintain network reachability information, alerting the local staff when the network was down and action needed to be taken to recover. Earlier, only a phone call from a rural clinic could alert the local administrator, and depending on the awareness of the staff at the rural clinic, this call would not always happen.

Second, kernel logs transferred using PhoneHome helped us diagnose several interesting problems. For example, in certain instances routers configured with two network interfaces reported only one interface as being active. Pairing this information with power data, we realized that a low voltage supply can prevent two radio interfaces from functioning simultaneously. In another instance, kernel logs and system messages allowed us to examine flash disk error messages and predict when disk partitions needed repartitioning or replacement.

Third, by examining the posted routing table and interface parameters, we were able to diagnose routing misconfigurations or badly assigned IP addresses.

Fourth, continuous monitoring of wireless link parameters helped us narrow the scope of the problems in many cases. Figure 7 shows the signal strength variation in some of our network links. While majority of these links show fairly stable signal strength, some of them show important variation over time. For example, a sudden 10dB signal drop on the link between Ambasam to Theni indicated some kind of a drastic event such as a possible antenna misalignment that needed an immediate visit. On the other hand, a steady decline in signal strength on the Bodi link indicated a gradual degradation of a connector

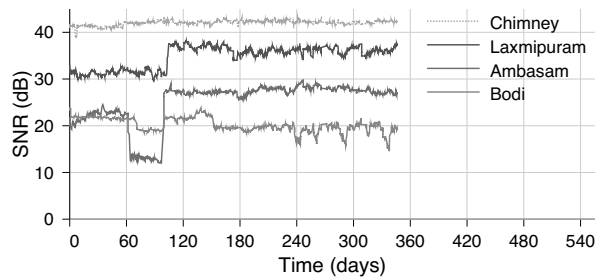


Figure 7: Signal strength (shown in dB) variation for all links. Each point is average of measurement over 2 days. The Ambasam link shows a temporary drop in SNR of 10 dB for about 40 days. While the Bodi link is gradually degrading as its SNR has dropped by 4 dB over the last year, the Chimney link's SNR has remained constant.

or the RF cable to the antenna, and required an eventual visit.

Tradeoffs: We contrast this with monitoring at AirJaldi where we use various off-the-shelf tools such as Nagios [10] and SmokePing [13] to collect node, link, and network level parameters. Information is stored at a local data server in Dharamsala and then copied to a US server for detailed analysis. Various graphing toolkits such as MRTG [25] are used to visualize trends and detect anomalies.

The difference in approach compared to Aravind is in part due to the higher experience of the AirJaldi staff, and in part due to the better connectivity we have to AirJaldi. The advantage of having local servers polling for information is that they can be configured by local staff to look for relevant problems, but such an approach is beneficial only if local staff are experienced enough to take advantage of these features.

After three years of operation, the local Aravind staff (some of whom we lost due to turnover after they gained more experience through our training) are more familiar with system configuration, and show less apprehension in taking the initiative and maintaining the system on their own. Therefore, we are now beginning to use a *pull-based* model.

In general, we believe that during the initial phase of a network deployment, minimal configuration *push-based* mechanisms are more appropriate for data collection. However, after building enough local expertise, the monitoring system should be migrated towards a more flexible *pull-based* approach.

5.2 Power

Power quality and availability has been our biggest concern at both Aravind and AirJaldi. Low-quality power damages the networking equipment (boards and power adapters) and sometimes also batteries. Over 90% of the

incidents we have experienced have been related to low power quality. Thus, designing to increase component reliability in the face of bad power is the most important task. We have developed two separate approaches to address the effects of low power quality. The first is a Low Voltage Disconnect (LVD) solution, which prevents both routers from getting wedged at low voltages and also over-discharge of batteries. The second is a low-cost power controller that supplies stable power to the equipment by combining input from solar panels, batteries, and even the grid.

Low Voltage Disconnect (LVD): Over-discharge of batteries can reduce their lifetime significantly. Owing to the poor quality of grid power, all AirJaldi routers are on battery backup. LVD circuits, built into battery chargers, prevent over-discharge of batteries by disconnecting the load (router) when the battery voltage drops below a threshold. As a beneficial side-effect, they prevent the router from being powered by a low-voltage source, which may cause it to hang. Off-the-shelf LVDs oscillated frequently, bringing the load up and down, and eventually damaging the board and flash memory. Every week, there were roughly fifty reboot incidents per router due to hangs caused by low voltage. However, we designed a new LVD circuit [24] with no oscillation and better delay; since then the hangs per week per router have reduced to near zero in the Dharamsala network.

Power Controller: We have developed a microcontroller-based solar power charge controller [31] that provides a stable input of 18 V to the routers and intelligently manages the charging and discharging of the battery pack. It has several features such as maximum power point tracking, low voltage disconnect, trickle charging and very importantly, support for remote management via ethernet. The setup is trivial as it supplies power to the router using PoE. This combination is novel for its price of around \$70.

We use TVS diodes to absorb spikes and surges and a robust voltage regulator to get clean 18V power from wide ranging input conditions. Figure 8 shows the flow of current through the board over a 60-hour period. First, we note that power is always available to the router. When enough sunlight is available, the solar panel powers the router and charges the battery. During periods of no sun, the battery takes over powering the router. The frequent swings observed on the left part of the graph are typical for a cloudy day. The graphs also demonstrate how the battery is continually charged when sunlight is available. We have measured a 15% more efficient power draw from the panels, and also expect that we can double battery life. Using the controller, we have not lost any routers from bad power, but it has been only 8 months of testing.

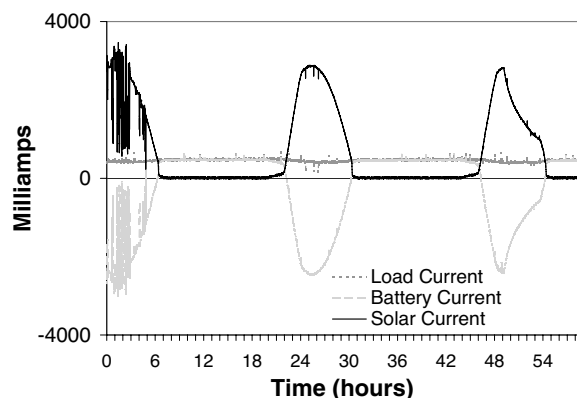


Figure 8: Current flow over 60 hours. The load stays even at 7W, while the solar panel and battery shift their relative generation over time. The battery current is negative when it is charging.

The controller reports solar panel, load and battery status information that can be used for remote diagnosis and some prediction of battery uptime and lifetime. A second version of the controller, currently under development, will add the feature to take grid-supplied power as input. This has two major advantages: the same setup can be used to stabilize grid power locally, and grid power can also be used to charge the batteries in addition to the solar power.

Tradeoffs: The real cost of power in rural areas is not just the raw grid electricity costs, but the cost of overcoming power availability and quality issues through UPS, battery-backups, and and chargers. The recurring costs can be quite high, and therefore solar power, although still expensive, becomes more competitive than expected as it can produce clean power directly. Currently we choose to use solar for very remote locations. At less remote and critical sites, we tend to use “dumb” analog chargers to reduce costs even further.

5.3 Backchannels

A wide variety of problems at Aravind and AirJaldi have caused link downtimes, leaving remote nodes disconnected. The failure of a single link makes part of the network unreachable although the nodes themselves might be functional. In many cases, if we had alternate access to the nodes, the fixes would have been simple such as correcting a router misconfiguration, or rebooting the router remotely. It is important to have out-of-band access or a backchannel to the nodes that is separate from the primary wireless path to it. Backchannel access is also useful in cases where the battery is discharging but the router is already down for other reasons. Information about the battery status from the charge controller via the backchannel would still be helpful. We have tried several approaches to backchannels in both networks.

Network Backchannel: At the Aravind Theni hospital, we already had some form of backchannel into the Theni network through VSAT. We use PhoneHome to open an SSH tunnel over the VSAT link through an HTTP proxy at the Aravind Madurai hospital. We configure PhoneHome to post monitoring data to our US-based server every 3 hours and also to open a reverse SSH tunnel through which we can log back in for administration purposes. Out of the 2300 posts expected from the router at Theni over 143 days (2 posts every 3 days), we only received 1510 of them, or about 65%. So this particular backchannel was not very reliable in practice, sometimes not working for long stretches of time. As a result, we used the solitary hospital laptop to connect directly to the Internet using a 1xRTT CDMA card to improve the availability of a backchannel into the network. However, this laptop was used for several other purposes (shared hardware is a common feature in rural areas) and was mostly unavailable. Furthermore, in many instances the network backchannel was not enough as the local wireless network would itself be partitioned.

Node Backchannel: At AirJaldi, we built a node backchannel mechanism using GPRS. In India at the moment, GPRS connectivity costs roughly \$10 per month for unlimited duration and bandwidth. We used a Netgear WGT634U router, interfaced through its USB 2.0 port with a mobile phone. The router runs PPP over GPRS and sets up an OpenVPN tunnel to a remote server. To enable remote diagnosis using this link, the backchannel router is connected to the main wireless router using ethernet and optional serial consoles. The backchannel router can also power-cycle the wireless router using a solid-state relay connected to one of its GPIO pins.

This approach has two advantages. First, the cellphone network is completely independent of the wireless link. Second, even though the mobile phone is charged from the same power source, it has its own battery which allows access via GPRS even if the main power source is down. However, for the Netgear router, we needed additional battery backup which adds to the maintenance complexity. One approach to simplify this setup for console access would be to use a Linux GPRS phone but we have not tried it yet.

Tradeoffs: Our experience with the GPRS backchannel in terms of providing real utility for system management has been mixed. Many common problems can be solved by alternative means in simpler ways. In cases of incorrect configuration of routers, we can imagine using the GPRS backchannel to fix problems. But at Aravind, when misconfigurations resulted in routing outages, we used cascaded hop-by-hop logins to move through the network, although this depended on at least the endpoint IP addresses to be set correctly. However, we can also

use Link Local IP addressing [12] to have independent hop-by-hop backchannels. Each link gets a local automatic IP address from a pre-assigned subnet that would work even when the system wide routing does not work. This can also be implemented by using virtual interfaces in the Atheros wireless driver [15]. Such virtual link configuration approaches could be permanent and also independent of any network configuration

We have also used the built-in WiFi radio of the backchannel netgear router to remotely scan local air interfaces for interferences or low RF signals from other routers, particularly after storms in Dharamsala. But we found the *most useful* feature of the GPRS backchannel to be console access to the router in case of failed attempts at remote firmware upgrades. But arguably, good practices of testing the upgrade locally on an identical router may suffice. This would mean reducing the variety of router platforms used in the field to standardize testing. However, this can be hard to do practically, especially in initial phases as rural networks move from pilots to scale. In future work we intend to continue exploring the idea of cellphone backchannels.

One idea is that instead of using GPRS as the backchannel, a cheaper mechanism could be using SMS channels. With SMS, console access would need to be implemented from scratch. Instead of console access, one approach would be to just query the remote router over SMS. The reply would have power parameters (grid power, remaining battery, voltage level of power supply), and basic status information from the wireless board if it is up. The phone would be connected to the router within the enclosure over serial. This is often feasible because many places have more ready access to SMS compared to GPRS. For example, all our rural clinics at Aravind, have some degree of SMS coverage provided by 2-3 providers at least.

5.4 Independent Recovery Mechanisms

Failure-independent recovery mechanisms are essential for managing systems remotely. The best solution is to have fully redundant systems, but they are often too expensive. An intermediate solution, more viable for rural areas, is to have some independent modules that enable diagnosis and some recovery (but not full functionality and so cannot do complete failover).

Alternate backchannels can enable independent access to various system components, and we include them in the design of independent recovery mechanisms. However in situations where the main router itself is wedged or is in a non-responsive state, we need components that can reset or reboot the main router for recovery. The components should not be affected by the failure themselves. In this section, we discuss software and hardware recovery.

Software watchdog: Essential software services can enter bad states and crash. For instance, we have seen wireless drivers have enter bad states that prevent the wireless card from receiving or transmitting packets even though the OS still keeps running. It is necessary to have a monitoring service that can either restart software services on the router or reboot the router itself.

We have built a software watchdog which is run by cron every 4 minutes. A configuration file lists what parameters to monitor such as IP reachability to a set of hosts, channel, SSID and BSSID changes, wireless operation mode as well as a list of processes that need to be running on the node. The configuration file also lists what actions to take upon failure of any of the tests, and how often a test is allowed to fail before an action is taken. Actions range from bringing the wireless interface down and up again, unloading and reloading kernel modules, to rebooting the node. We use this software watchdog in the AirJaldi network currently.

Hardware watchdog: An on-board hardware watchdog will reboot the router periodically unless it gets reset periodically after receiving keep-alive messages from the router. This is a vital feature, but most of the low-cost routers used at AirJaldi do not actually have on-board watchdogs. To address this we have designed for \$0.25, a simple external hardware watchdog (a simple delay circuit) that interfaces with the board's GPIO line. We have designed this watchdog to plug into the router's power input port and to also accept PoE-enabled power so it can also power PoE-less routers, which allows us to use lower-cost routers as well. All the boards we use at Aravind have on-board watchdogs, but if the board is wedged due to lower voltage, then the watchdog itself will be rendered useless. However, we can avoid this by using the LVDs we have designed. In some cases, we are also using the power controller described in Section 5.2 as a form of external hardware watchdog; it monitors the board over ethernet and power-cycles it via PoE if it does not hear a keep-alive message in time.

Enabling Safe Fallback: As future work, we intend to use the backchannel and the independent recovery plane to implement *safe fallback mechanism* for upgrades. When upgrading the OS on a wireless router, we could use a software watchdog that will be configured to check that the upgrade does not violate any required properties. For example, the board should be able to initialize all the drivers, and ping local interfaces and remote nodes as well. If these are not satisfied, we should go back to a previously known fail-safe OS state. This can be combined with a hardware watchdog mechanism that can reboot the router to a fail-safe OS state in cases where the newly installed OS does not even boot.

5.5 Software Design

We have written substantial software for the WiLDNet MAC, monitoring, logging, remote management, fault diagnosis, and fault prediction. In this section we focus on aspects that we have not previously discussed: the boot loader, and configuration and status tools. Both play an important role in reducing failures.

Read-only Operating System: We saw at the Aravind network that the CF cards used in the wireless routers would often get corrupted because of frequent and unexpected reboots. Writing even a single bit of data can corrupt a flash disk. We discovered at AirJaldi that if an oscillating LVD keeps rebooting a router, some write to the CF card during boot up will eventually fail and corrupt the flash. Unfortunately, since most boot loaders write to flash during the boot up process, we had to replace the boot loader with our own version that does not perform any writes at all.

In addition, it is better to mount the main OS partition read-only so that no write operations occur throughout the normal life cycle of the router. For log collection, we have an extra read-write partition on the CF card. However, in production systems, it would be preferable to have all the partitions to be read-only mounted.

Configuration and Status Tools: To train local staff in the administration of wireless network without exposing them to the details of underlying Linux configuration files, we designed a web-based GUI for easy configuration and display of simple status information about a particular router.

But to further aid local staff in diagnosing problems we need to build tools that can present an easy to understand view of the problem. For example, a simple mechanism at vision centers can indicate (via something as simple as LEDs) that the local wireless router is up and running, but that reachability to the remote router is down. This will minimize the tendency of *self-fixing* where local staff unnecessarily try to modify the local setup without realizing that the problem might be elsewhere.

6 Related Work

WiFi-based deployments: There have been several development projects that use WiFi-based network connectivity for applications such as healthcare (Ashwini [4]), the Digital Gangetic Plains [8]), e-literacy and vocational training (the Akshaya network [2]), education (CRC-Net [7]) and so on. However, our deployment is possibly the first that takes a systematic approach towards sustainability and both projects are in active use by thousands of users. There are a number of community wireless projects in the US ([5, 6, 11]) that use a combination of open source monitoring tools, but they focus on

a smaller range of operational challenges. Raman et al. in [30] try to summarize all the open issues in deploying rural wireless such as network planning, protocols, management, power and applications but they mainly focus on modifying the MAC and conserving power using *Wake-on-LAN* [23] techniques.

Long distance point-to-point WiFi: Given the cost and performance promises of 802.11 rural connectivity, there have been several efforts to analyze the behavior [19, 33] and improve the performance of multi-hop long-distance WiFi networks, including the design of an TDMA-based MAC layer [29] that relies on burst synchronization to avoid interference, and channel allocation policies to maximize network throughput [28]. Our work [26] builds and improves on these efforts, delivering a real-world implementation that delivers high-performance (5-7Mbps for links up to 382 Km), predictable behavior, and flexibility to accommodate various types of traffic. Raman *et al.* [32] also investigate network planning solutions that minimize costs by optimizing across the space of possible network topologies, tower heights, antenna types and alignment and power assignments.

Long distance point-to-multipoint WiFi: It is not always possible to design a network with just point-to-point links. For example, in topologies where there is not much angular separation between clients with respect to a central location, it is infeasible to have separate point-to-point links to each client using directional antennas. Instead, an interesting compromise is to use sector antennas where some nodes run a point-to-multipoint (PMP) MAC protocol to provide access to a large number of clients that do not have very high individual throughput requirements while the long distance links still use the point-to-point MAC protocol [27, 18]. We are currently in the process of extending the WiLDNet MAC protocol to support point-to-multipoint configurations as well.

Remote management: There has been a lot of work on remote operation and upgrades to large-scale datacenters [14, 17] that have reliable power and network connectivity. There has also been work on online software upgrades to sensor networks [21]. However remote management solutions for wireless networks that located in remote rural regions has not received a lot of attention. In this spectrum, Meraki [9] provides a remote management suite for WiFi networks where all the monitoring, configuring, diagnosis and periodic updates for their field-deployed routers is hosted on the Meraki server.

7 Conclusion

We presented a wide range of operational challenges from three years of deployment experience with two different rural wireless networks. Although work to date

Type of problem	Instances	Recovery time	Who solved it	Who solves it now
Circuit breaker trip at node locations	S:26 V:33 C:4	1 day	Staff: Flip the breaker physically at location, added UPS	Staff: Monitoring system triggers that node is down
PoE stopped working (transformer explosion)	1	1-7 days	Integrators: Replaced PoE	Staff: Replace PoE by checking connectivity and components
Loose ethernet cable jacks	M:12 C:2 T:7	1-7 days	Experts, Staff: Re-crimp RJ-45 with help from experts, train staff to check for loose cables	Staff: Monitoring system triggers that wireless link is up but ethernet is down
Routing misconfiguration: incorrect static routes, absent default gateway	Routing:2 Gateway:4	1-7 days	Experts: Using reverse SSH tunnel Integrators: Using config tool	Staff/Integrators: Use config tool for routing
CF card corruption: disk full errors	Replace:2 Fix:10		Integrators, Staff: Replace CF card Experts: Run fsck regularly	Automatic: Run fsck on problem Staff: Replace CF cards after config.
Wall erected in front of antenna: link went down	1	2 months	Experts: After physical verification	Staff: Ensure line of sight
Ethernet port on board stopped working	M:2	N/A	Integrators: Replace router board	Staff/Integrators: Replace boards

Table 3: List of failures that have occurred since January 2005 at various locations in the Aravind network. For each fault, we list the downtimes, and who among **staff**, **integrators**, or remote **experts** used to solve the problem, and who solves it now. This information has been collated from logs and incident reports maintained by the local administrators and remote experts respectively. It is an underestimate as not all failures are accounted for in the local logs maintained by local staff.

largely focuses on performance, the primary obstacle to real impact for these networks is keeping them alive over the long term. Based on our experiences, we conclude by summarizing three broad lessons which we believe apply to other projects in developing regions.

Prepare for absence of local expertise: Most projects assume that training will solve the need for local IT staff, but this is quite difficult. Although we have had some success with this at AirJaldi, it is limited due to high staff turnover. In some sense, better training leads to higher turnover. So instead, we have worked to reduce the need for highly trained staff on multiple levels.

Starting at the lowest layers, we have pushed hard on improving the quality of power and the ability of nodes to reboot themselves into a known good state. We have added substantial software for self validation, for data collection and monitoring. We also developed support for remote management, although it is limited by connectivity issues, especially during faults; in turn, we looked at backchannels to improve the reach for remote management. We also developed GUI tools that are much easier for local staff to use and that are intended to be educational in nature. At the highest level, the network integrators step in to handle issues that local staff cannot solve; earlier local staff would wait until we solved the problem, resulting in extended downtimes. This transition is shown in the partial list of failures in Table 3 from the Aravind network. For each fault we indicate how it was solved initially, what the associated downtime was, and also how that same fault is being solved now.

Redesign of components is oftens enough: As mentioned earlier in Section 4, because of harsh environmental conditions and unreliable power, commodity components fail more often in rural areas. One solution is to use expensive equipment such as military grade routers and big battery backups or diesel generators, as is done with cellular base stations at great cost. However, we aim to use low-cost commodity hardware for affordability.

In practice, even simple redesign of selected hardware components can significantly decrease the failure rates without adding much cost. In addition to getting WiFi to work for long distances, we also developed software and hardware changes for low-voltage disconnect, for cleaner power, and for more reliable automatic reboots, and we developed better techniques to avoid damage due to lightning and power surges.

The real cost of power is in cleaning it up: The key is to understand that the real cost of power in rural areas is not the cost of grid power supply, but of cleaning it using power controllers, batteries and solar-power backup solutions. Some development projects incorrectly view the cost of electricity as zero, since it is relatively common to steal electricity in rural India.¹ However, the grid cost is irrelevant for IT projects, which generally need clean power (unlike lighting or heating). Due to short lifetimes of batteries and ineffective UPSs, power cleaning is a recurring cost. Solar power, although still expensive, is thus more competitive than expected as it produces clean power directly. We currently use solar power for relays or other locations where power is not available, and try

to manage grid power elsewhere. At the same time, it is critical to improve the tolerance for bad power of all of the equipment, and to plan for sufficient back up power.

In the end, there remains much to do to make these networks easier to manage by the local staff; progress is required on all fronts. However, even the changes implemented so far have greatly reduced the number of failed components, have increased the ability of local staff to manage network problems, and have helped to grow the networks without significantly growing the staff. Both networks are not only helping thousands of real users, but are also experiencing real growth and increased impact over time.

Acknowledgments

We would like to thank the Aravind Eye Care System, the AirJaldi Community Network, Ermanno Pietrosevoli, and Alan Mainwaring for their help. We would also like to thank our shepherd, Robert Morris, for his contributions in improving the paper, and our reviewers, for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. 0326582.

References

- [1] AirJaldi Wireless Network. <http://summit.airjaldi.com>.
- [2] Akshaya E-Literacy Project. <http://www.akshaya.net>.
- [3] Aravind Eye Care System. <http://www.aravind.org>.
- [4] Ashwini: Association for Health Welfare in the Nilgiris. <http://www.ashwini.org>.
- [5] Bay Area Research Wireless Network. <http://www.barwn.org>.
- [6] Champaign-Urbana Community Wireless Network. <http://www.cuwin.net>.
- [7] CRCNet: Connecting Rural Communities Using WiFi. <http://www.crc.net.nz>.
- [8] Digital Gangetic Plains. <http://www.iitk.ac.in/mladgp/>.
- [9] Meraki Wireless Mesh Routers. <http://www.meraki.net>.
- [10] Nagios Wireless Monitoring. <http://www.nagios.org>.
- [11] NY Wireless Network. <http://www.nycwireless.net>.
- [12] RFC 3927: Dynamic Configuration of IPv4 Link-Local Addresses. <http://www.ietf.org/rfc/rfc3927.txt>.
- [13] SmokePing. <http://oss.oetiker.ch/smokeping/>.
- [14] S. Ajmani, B. Liskov, and L. Shriram. Scheduling and simulation: How to upgrade distributed systems. In *HotOS-IX*, 2003.
- [15] Atheros. MadWiFi driver for Atheros Chipsets. <http://sourceforge.net/projects/madwifi/>.
- [16] P. Bhagwat, B. Raman, and D. Sanghi. Turning 802.11 Inside-out. In *Hotnets-III*, 2004.
- [17] E. Brewer. Lessons from Giant-scale Services. *IEEE Internet Computing*, 2001.
- [18] K. Chebrolu and B. Raman. FRACTEL: A Fresh Perspective on (Rural) Mesh Networks. In *ACM SIGCOMM Workshop on Networked Systems for Developing Regions (NSDR)*, August 2007.
- [19] K. Chebrolu, B. Raman, and S. Sen. Long-Distance 802.11b Links: Performance Measurements and Experience. In *ACM MOBICOM*, 2006.
- [20] M. Gregory. India Struggles with Power Theft. <http://news.bbc.co.uk/2/hi/business/4802248.stm>, 2006.
- [21] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI*, 2004.
- [22] Marratech. Videoconferencing Software. <http://www.marratech.com>.
- [23] N. Mishra, K. Chebrolu, B. Raman, and A. Pathak. Wake-on-WLAN. In *WWW*, May 2006.
- [24] P. Narhi and Y. Ben-David. Air Jaldi Charger Hardware Design. http://drupal.airjaldi.com/system/files/Jaldi_Charger_design_1.6.3.pdf.
- [25] T. Oetiker. MRTG: The Multi Router Traffic Grapher. <http://oss.oetiker.ch/mrtg/>.
- [26] R. Patra, S. Nedeveschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks. In *NSDI*, 2007.
- [27] K. Paul, A. Varghese, S. Iyer, and B. R. A. Kumar. WiFiRe: Rural Area Broadband Access Using the WiFi PHY and a Multisector TDD MAC. *New Directions in Networking Technologies in Emerging Economics, IEEE Communications Magazine*, 2006.
- [28] B. Raman. Channel Allocation in 802.11-based Mesh Networks. In *IEEE INFOCOM*, April 2006.
- [29] B. Raman and K. Chebrolu. Design and Evaluation of a new MAC Protocol for Long-Distance 802.11 Mesh Networks. In *ACM MOBICOM*, August 2005.
- [30] B. Raman and K. Chebrolu. Experiences in using WiFi for Rural Internet in India. *IEEE Communications Magazine*, January 2007.
- [31] M. Ramos and E. Brewer. TIER Solar Controller. <http://tier.cs.berkeley.edu/wiki/Power>.
- [32] S. Sen and B. Raman. Long Distance Wireless Mesh Network Planning: Problem Formulation and Solution. In *WWW*, 2007.
- [33] A. Sheth, S. Nedeveschi, R. Patra, S. Surana, L. Subramanian, and E. Brewer. Packet Loss Characterization in WiFi-based Long Distance Networks. In *IEEE INFOCOM*, 2007.
- [34] L. Subramanian, S. Surana, R. Patra, M. Ho, A. Sheth, and E. Brewer. Rethinking Wireless for the Developing World. In *Hotnets-V*, 2006.
- [35] S. Surana, R. Patra, and E. Brewer. Simplifying Fault Diagnosis in Locally Managed Rural WiFi Networks. In *ACM SIGCOMM Workshop on Networked Systems for Developing Regions (NSDR)*, 2007.

Notes

¹The tolerance of theft is a kind of subsidy for the poor, but it is badly targeted as others steal power too. India loses about 42% of its generated electricity to a combination of theft and transmission losses (vs. 5–10% in the US) [20].

UsenetDHT: A low-overhead design for Usenet

Emil Sit, Robert Morris, and M. Frans Kaashoek
MIT CSAIL

Abstract

Usenet is a popular distributed messaging and file sharing service: servers in Usenet flood articles over an overlay network to fully replicate articles across all servers. However, replication of Usenet's full content requires that each server pay the cost of receiving (and storing) over 1 Tbyte/day. This paper presents the design and implementation of UsenetDHT, a Usenet system that allows a set of cooperating sites to keep a shared, distributed copy of Usenet articles. UsenetDHT consists of client-facing Usenet NNTP front-ends and a distributed hash table (DHT) that provides shared storage of articles across the wide area. This design allows participating sites to partition the storage burden, rather than replicating all Usenet articles at all sites.

UsenetDHT requires a DHT that maintains durability despite transient and permanent failures, and provides high storage performance. These goals can be difficult to provide simultaneously: even in the absence of failures, verifying adequate replication levels of large numbers of objects can be resource intensive, and interfere with normal operations. This paper introduces Passing Tone, a new replica maintenance algorithm for DHash [7] that minimizes the impact of monitoring replication levels on memory and disk resources by operating with only pairwise communication. Passing Tone's implementation provides performance by using data structures that avoid disk accesses and enable batch operations.

Microbenchmarks over a local gigabit network demonstrate that the total system throughput scales linearly as servers are added, providing 5.7 Mbyte/s of write bandwidth and 7 Mbyte/s of read bandwidth per server. UsenetDHT is currently deployed on a 12-server network at 7 sites running Passing Tone over the wide-area: this network supports our research laboratory's live 2.5 Mbyte/s Usenet feed and 30.6 Mbyte/s of synthetic read traffic. These results suggest a DHT-based design may be a viable way to redesign Usenet and globally reduce costs.

1 Introduction

For decades, the Usenet service has connected users world-wide. Users post articles into newsgroups which are propagated widely by an overlay network of servers. Users host lively discussions in newsgroups and also, because articles can represent multi-media files, cooperatively produce a large shared pool of files. A major at-

traction of Usenet is the incredible diversity and volume of content that is available.

Usenet is highly popular and continues to grow: one Usenet provider sees upwards of 40,000 readers reading at an aggregate 20 Gbit/s [35]. Several properties contribute to Usenet's popularity. Because Usenet's design [1, 19] aims to replicate all articles to all interested servers, any Usenet user can publish highly popular content without the need to personally provide a server and bandwidth. Usenet's maturity also means that advanced user interfaces exist, optimized for reading threaded discussions or streamlining bulk downloads. However, providing Usenet service can be expensive: users post over 1 Tbyte/day of new content that must be replicated and stored.

This paper presents UsenetDHT, a system that allows a group of cooperating servers to share the network and storage costs of providing Usenet service. Such an approach benefits both operators and users of Usenet: operators can use the savings from UsenetDHT to provide better service (for example, save articles for longer), or pass on reduced costs to users. With costs lower, more organizations may be able to afford to provide Usenet and make Usenet available to more users.

UsenetDHT uses a distributed hash table (DHT) to store and maintain Usenet articles across participating sites [32]. A DHT provides a single logical storage system for all sites; it handles data placement, load balance and replica maintenance. Front-ends speaking the standard news protocol accept new articles and store them into the common DHT. The front-ends then flood information about the existence of each article to all other front-ends in the UsenetDHT deployment. Instead of transmitting and storing n copies of an article (one per server), UsenetDHT initially stores only two for the entire deployment; the extra copy allows for recovery from failure. Thus, per server, UsenetDHT reduces the load of receiving and storing articles by a factor of $O(n)$.

To service reads, UsenetDHT front-ends must read from the DHT. The front-ends employ caching to ensure that each article is retrieved at most once for local clients. Thus, even if clients read all articles, UsenetDHT never creates more copies than Usenet. Further, statistics from news servers at MIT indicate that, in aggregate, clients read less than 1% of the articles received [29, 36]. If MIT

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, <http://project-iris.net/>.

partnered with similar institutions, UsenetDHT could reduce its costs for supporting Usenet substantially.

In addition to delivering cost savings, UsenetDHT is incrementally deployable and scalable. UsenetDHT preserves the NNTP client interface, allowing it to be deployed transparently without requiring that clients change their software. The use of an underlying DHT allows additional storage and front-end servers to be easily added, with the DHT handling the problem of re-balancing load. Additionally, UsenetDHT preserves the existing economic model of Usenet. Compared to alternative content distribution methods such as CDNs or `ftp` mirror servers that put the burden of work and cost on the publisher, UsenetDHT leaves the burden of providing bandwidth at the Usenet front-end, close to the consumer.

Despite the advantages of the UsenetDHT design, the workload of Usenet places significant demands on a DHT. The DHT must support sustained high throughput writes of multiple megabytes per second. Further, since even MIT's fractional feed of 2.5 Mbyte/s generates 350 Gbyte/day of replicas, unless a new disk is acquired every day, the system will run continuously at disk capacity, necessitating continuous object deletions to reclaim space.

The DHT that UsenetDHT uses must also provide data availability and durability. This is achieved through replica maintenance, where a DHT replaces lost replicas to prevent data loss or unavailability. The goal of replica maintenance is to avoid losing the last replica, but without making replicas unnecessarily, since objects are expensive to copy across the network. To achieve this, a maintenance algorithm must have an accurate estimate of the number of replicas of each object in order to avoid losing the last one. Since DHTs partition responsibility for maintenance across servers, a simple solution would have each DHT server check periodically with other servers to determine how many replicas exist for objects in its partition. However, naïvely exchanging lists of object identifiers in order to see which replicas are where can quickly become expensive [16].

A more efficient mechanism would be to simply exchange lists once and then track updates: a synchronization protocol (e.g., [4, 22]) can identify new objects that were added since the last exchange as well as objects that have been deleted to reclaim space. However, this alternative means that each server must remember the set of replicas held on each remote server that it synchronizes with. Further, each synchronization must cover objects inserted in well-defined periods so that when a server traverses its local view of replicas periodically to make repair decisions, it considers the same set of objects across all servers. These problems, while solvable, add complexity to the system. The storage and traversal of such sets can also cause disk activity or memory pressure, interfer-

ing with regular operations.

To handle maintenance under the workload of Usenet, we introduce *Passing Tone*, a maintenance algorithm built on Chord and DHash [7]. Instead having each object's successor ensure that sufficient replicas exist within the successor list, Passing Tone has all servers in the object's successor list ensure that they replicate the object. Each Passing Tone server makes maintenance decisions by synchronizing alternately with its successor and predecessor against the objects it stores locally. Synchronization identifies objects that the server should replicate but doesn't. These objects are pulled from its neighbors. In this way, Passing Tone ensures that objects are replicated on the successor list without having to explicitly track/count replicas, and without having to consider a consistent set of objects across several servers. As a result, Passing Tone can efficiently maintain replicas in a system where constant writes produce large numbers of objects and the need to continuously delete expired ones.

Our implementation of Passing Tone lays out data structures on disk efficiently to make writes, deletions, and synchronization operations efficient. We deployed this implementation on 12 wide-area servers, and this deployment handles the live 2.5 Mbyte/s Usenet feed received at MIT. The deployment can support an aggregate read throughput of 30.6 Mbyte/s from wide-area clients. Benchmarks run over our local gigabit network show that the total read and write capacity scale as servers are added.

This paper makes three contributions: it presents the design of UsenetDHT, a system that reduces the individual cost of operating a Usenet server for n participants by a factor of $O(n)$ through the use of a DHT; it introduces the Passing Tone algorithm that provides efficient maintenance for DHTs for workloads with many concurrent write and expiration operations; and, it demonstrates that an implementation of UsenetDHT can support MIT's Usenet feed and should scale to the full feed.

The rest of this paper is structured as follows. Section 2 describes Usenet and briefly characterizes its traffic load. UsenetDHT's design is presented in Section 3. Section 4 describes the Passing Tone algorithm. Section 5 discusses the Passing Tone implementation, which is then evaluated in Section 6. Finally, Section 7 presents related work and we conclude in Section 8.

2 Usenet background

Usenet is a popular distributed messaging service that has been operational since 1981. Over the years, its use and implementation has evolved. Now people use Usenet in two primary ways. First, users participate in discussions about specific topics, which are organized in newsgroups. The amount of traffic in these text groups has been relatively stable over the past years. Second, users post binary articles (encoded versions of pictures, audio

files, and movies). This traffic is increasing rapidly because Usenet provides an efficient way for users to distribute large multi-media files. Users can upload articles to Usenet once and then Usenet takes charge of distributing the articles.

Usenet distributes articles using an overlay network of servers that are connected in a peer-to-peer topology. Servers are distributed world-wide and each server peers with its neighbors to replicate all articles that are posted to Usenet. The servers employ a flood-fill algorithm using the NetNews Transfer Protocol (NNTP) to ensure that all articles reach all servers [1, 19].

As a server receives new articles (either from local posters or its neighbors), it floods NNTP CHECK messages to all its other peers who have expressed interest in the newsgroup containing the article. If the remote peer does not have the message, the server feeds the new article to the peer with the TAKETHIS message. Because relationships are long-lived, one peer may batch articles for another when the other server is unavailable, but today's servers typically stream articles to peers in real-time.

The size of Usenet is hard to measure as each site sees a different amount of traffic, based on its peering arrangements. An estimate from 1993 showed an annual 67% growth in traffic [34]. Currently, in order to avoid missing articles, top servers have multiple overlapping feeds, receiving up to 3 Tbyte of traffic per day from peers, of which approximately 1.5 Tbyte is new content [10, 12]. Growth has largely been driven by increased postings of binary articles. The volume of text articles has remained relatively stable for the past few years at approximately 1 Gbyte of new text data, from approximately 400,000 articles [14]. As a result of the large volume of traffic, providers capable of supporting a full Usenet feed have become specialized. Top providers such as `usenetserver.com` and GigaNews have dedicated, multi-homed data centers with many servers dedicated to storing and serving Usenet articles.

A major differentiating factor between Usenet providers is the degree of article *retention*. Because Usenet users are constantly generating new data, it is necessary to *expire* old articles in order to make room for new ones. Retention is a function of disk space and indicates the number of days (typically) that articles are kept before being expired. The ability to scale and provide high performance storage is thus a competitive advantage for Usenet providers as high retention allows them to offer the most content to their users. For example, at the time of this writing, the longest retention available is 200 days, requiring at least 300 Tbyte of data storage.

The read workload at major news servers can be extremely high: on a weekend in September 2007, the popular `usenetserver.com` served an average of over 40,000 concurrent clients with an aggregate bandwidth of

20 Gbit/s [35]. This suggests that clients are downloading continuously at an average of 520 Kbit/s, most likely streaming from binary newsgroups.

Usenet's economics are structured to allow providers to handle the high costs associated with receiving, storing, and serving articles. Perhaps surprisingly in the age of peer-to-peer file sharing, Usenet customers are willing to pay for access to the content on Usenet. Large providers are able to charge customers in order to cover their costs and produce a profit. Unfortunately, universities and other smaller institutions may find it difficult to bear the cost of operating an entire Usenet feed. UsenetDHT is an approach to bring these costs down and allow more sites to operate Usenet servers.

3 UsenetDHT design

UsenetDHT targets mutually trusting organizations that can cooperate to share storage and network load. Prime examples of such organizations are universities, such as those on Internet2, that share high-bandwidth connectivity internally and whose commercial connectivity is more expensive. For such organizations, UsenetDHT aims to:

- reduce bandwidth and storage costs in the common case for all participants;
- minimize disruption to users by preserving an NNTP interface; and
- preserve the economic model of Usenet, where clients pay for access to their local NNTP server and can publish content without the need to provide storage resources or be online for the content to be accessible.

UsenetDHT accomplishes these goals by replacing the local article storage at each NNTP server with shared storage provided by a distributed hash table (DHT). A front-end that speaks the standard client transfer protocol (NNTP) allows unmodified clients access to this storage at each site.

3.1 Design overview

Each article posted to Usenet has metadata—header information such as the subject, author, and newsgroups—in addition to the article itself. Articles entering a UsenetDHT deployment (for example, from a traditional Usenet feed or a local user) will come with metadata and the article bundled together. UsenetDHT floods the metadata among its participating peers in the same way as Usenet does. UsenetDHT, however, stores the articles in a DHT.

In UsenetDHT, each site contributes one or more servers with dedicated storage to form a DHT, which acts like a virtual shared disk. The DHT relieves UsenetDHT from solving the problem of providing robust storage; DHT algorithms deal with problems of data placement, maintenance in the face of failures, and load balance as the number of servers and objects in the system

increases [5, 33]. To support a full feed, each server in a homogeneous deployment need provide only $O(1/n)$ -th of the storage it would need to support a full feed by itself.

NNTP front-ends store incoming articles into the DHT using `put` calls; these articles may come from local users or from feeds external to the UsenetDHT deployment. To send articles upstream to the larger Usenet, front-ends in a UsenetDHT deployment have the option of arranging a direct peering relationship with an external peer or designating a single front-end to handle external connectivity.

Usenet is organized into newsgroups; when an article is posted, it includes metadata in its headers that tells the NNTP front-ends which groups should hold the article. In UsenetDHT, each NNTP front-end receives a copy of all headers and uses that information to build up a mapping of newsgroups to articles stored to local disk for presentation to its local clients. In particular, each front-end keeps an article index independently from other sites. UsenetDHT does not store group lists or the mapping from newsgroups to articles in the DHT.

Distributing metadata to all sites has several advantages. First, it guarantees that a site will be able to respond to NNTP commands such as `LIST` and `XOVER` without consulting other front-ends. These commands are used by client software to construct and filter lists of articles to present to actual users, before any articles are downloaded and read. Second, it leaves sites in control over the contents of a group as presented to their local users. In particular, it allows sites to have different policies for filtering spam, handling moderation, and processing cancellations.

Clients access UsenetDHT through the NNTP front-ends. When a user reads an article, the NNTP front-end retrieves the article using a DHT `get` call and caches it. Local caching is required in order to reduce load on other DHT servers in the system and also to ensure that UsenetDHT never sends more traffic than a regular Usenet mesh feed would. If sites cache locally, no DHT server is likely to experience more than n remote read requests for the DHT object for a given article. This cache can also be shared between servers at a site. Each site will need to determine an appropriate cache size and eviction policy that will allow it to serve its readership efficiently.

3.2 Write and read walk-through

To demonstrate the flow of articles in UsenetDHT more precisely, this section traces the posting and reading of an article. Figure 1 summarizes this process.

A news reader posts an article using the standard NNTP protocol, contacting a local NNTP front-end. The reader is unaware that the front-end is part of UsenetDHT, and not a standard Usenet server. Upon receiving the posting, the UsenetDHT front-end uses `put` to insert the article in the DHT. In the `put` call, the front-end uses the SHA-1

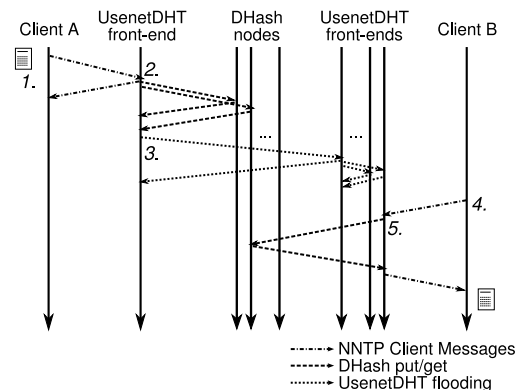


Figure 1: Messages exchanged during during UsenetDHT reads and writes. 1. Client A posts an article to his local NNTP front-end. 2. The front-end stores the article in DHash (via a DHash gateway, not shown). 3. After successful writes, the front-end propagates the article's metadata to other front-ends. 4. Client B checks for new news and asks for the article. 5. Client B's front-end retrieves the article from the DHT and returns it to her.

hash of the article's content as the key: since DHash partitions data across servers by the key, using a hash function ensures articles will be distributed evenly across the participating servers. By providing this key in a `get` call, any UsenetDHT front-end can retrieve the article from the DHT. The use of a content-hash key also allows front-ends to detect data corruption by verifying the integrity of data received over the network.

After the article has been successfully stored in the DHT, the front-end propagates the article to its peers using a `TAKEDHT` NNTP message. This message is similar to the standard `TAKETHIS` message but only includes the header information and the content-hash key (as an `X-ChordID` header). This information is sufficient for the peers to insert the article into their local group indices, provide a summary of the article to readers connecting to the front-end and retrieve the contents of the article when a reader requests it. Each front-end, upon receiving the article, also shares the announcement with its other peers. In this manner, the article's existence is eventually flooded to all front-ends in the deployment.

When a user wishes to read a newsgroup, his news reader software requests a list of new articles from his UsenetDHT front-end. The front-end responds with a summary of articles that it has accumulated from its peers. This summary is used by the reader to construct a view of the newsgroup. When the client requests an article body, the front-end first checks its local cache; if the article is not present, it calls `get`, supplying the key for the article as argument. When the front-end obtains the article data

from the DHT, it inserts the article into the cache and returns the article to the reader. As with posting, the reader is unaware that the news server is part of UsenetDHT.

3.3 Expiration

UsenetDHT will insert dozens of objects into the DHT per second, resulting in millions of objects per day. After an initial start-up period, the DHT will be operating at full storage capacity. Thus, some mechanism is needed to delete older objects to make room for newer ones.

Usenet servers have long used *expiration times* to bound the lifetime of articles. Each deployment of UsenetDHT sets a single common expiration policy across all participants. This policy can vary according to the type of newsgroup (e.g., preserving text discussions for longer than binary articles). A common policy is required to ensure that the list of articles for a given group at any UsenetDHT front-end will accurately reflect the articles that are available in the underlying DHT.

3.4 Discussion

UsenetDHT converts a fully decentralized system based on pushing content to all servers into a partially decentralized one, where individual front-ends pull the content from their peers. Thus, servers must participate in processing DHT lookups for all articles, even for readers at other sites. Conversely, each server depends on other servers to respond to its read requests. This motivates requiring a trusted set of participating sites.

A DHT-based design also represents a sacrifice in terms of site control. Sites lose control over what articles are stored on and transit their servers. While sites retain the ability to specify which newsgroups are indexed locally, they must participate fully in the DHT and store articles that are in groups that they do not make available to local clients. Policies of content filtration, handled in regular Usenet by filtering the list of newsgroups that are peered to a site, must now be handled per deployment. Future work may address this issue with the use of sub-rings [20].

The key benefit of the DHT approach is that each site will receive article data proportional to the amount of storage they contribute to the system, rather than a complete copy of the feed. UsenetDHT reduces the cost of receiving and storing a Usenet feed at each site by a factor of $O(n)$ (assuming equal amounts of storage contributed by each site) by eliminating the need to massively replicate articles. Our UsenetDHT implementation replicates articles for durability at two servers, so the cost of receiving and storing articles is reduced by a factor of $n/2$.

UsenetDHT employs local caching to use no more bandwidth than standard Usenet in the worst case where every article is read at every server. Fortunately, at the organizations we target, it is unlikely that all articles will be read, resulting in substantial savings per site overall.

As a result, total storage can be reduced or re-purposed to provide additional retention instead of storing replicas.

The flooding network used by UsenetDHT to propagate metadata follows peering relationships established by the system administrators. A reasonable alternative may be to construct a distribution tree automatically [17]. An efficient broadcast tree would reduce link stress by ensuring that data is transmitted over each link only once. However, the metadata streams are relatively small and operators may prefer to avoid the complexity and possible fragility involved in deploying such an overlay.

4 Passing Tone

The DHT storing articles for UsenetDHT needs an efficient maintenance algorithm that provides high availability and durability for objects: maintenance ensures that a sufficient number of replicas exists to prevent object loss due to server failures. This section introduces the Passing Tone algorithm; Passing Tone maintains replicated objects through a user-specified expiration time while reducing the number of disk seeks and memory required to make maintenance decisions.*

We present Passing Tone in the context of the DHash DHT. Each server in DHash has a single database that is shared across all of its virtual servers. DHash stores object replicas initially on the first k successors of the object's key; this set of servers is called the *replica set*. Passing Tone ensures that all members of the replica set have a replica of the object, despite the fact that this set changes over time. The value of k is set system-wide and affects the ability of the system to survive simultaneous failures; the reader is referred to our prior work for more on the role of k [5]. For UsenetDHT, $k = 2$ is sufficient.

4.1 Challenges

The main challenge in the design of Passing Tone is balancing the desire to minimize bandwidth usage (e.g., by not repeatedly exchanging object identifier lists and not creating too many replicas) with the need to avoid storing and updating state about remote servers. This problem largely revolves around deciding what information to keep on each server to make identifying objects that need repair easy and efficient.

The ideal maintenance algorithm would only generate repairs for those objects that have fewer than k remaining replicas. Such an algorithm would minimize replica movement and creation. A straightforward approach for achieving this might be to track the location of all available replicas, and repair when k or fewer remain. In a peer-to-peer system, where writes are sent to individual servers without going through any central host, tracking

*The name "Passing Tone" draws an analogy from the musical meaning "Chord" to the action of the maintenance algorithm: passing tones are notes that pass between two notes in a chord.

replica locations would require that each server frequently synchronize with other servers to learn about any new updates. The resulting replica counts would then periodically be checked to identify objects with too few replicas and initiate repairs.

The difficulty with this approach lies in storing state about replica locations in a format that is easy to update as objects are added and deleted but also easy to consult when making repair decisions. For ease of update and to allow efficient synchronization, information about replicas on each server needs to be kept in a per-server structure. However, to make repair decisions, each server requires a view of replicas over all servers to know how many replicas are currently available. While these views can be derived from one another, with millions of objects across dozens of servers, it is expensive to construct them on the fly. For example, Merkle tree construction is CPU intensive and would require either random disk I/O or use significant memory. Storing both views is possible but would also require random disk I/O that would compete with regular work.

Worse, even if state about replica locations can be efficiently stored, continuous writes and deletions means that any local state quickly becomes stale. Unfortunately, if information about objects on a remote server is stale—perhaps because an object was written to both the local and remote server but the local and remote have not yet synchronized—the local server may incorrectly decide to generate a new replica. Such *spurious repairs* can be expensive and are hard to avoid.

4.2 Passing Tone overview

Passing Tone deals with these challenges by removing the need to track object locations explicitly. Instead each server in Passing Tone:

- only keeps a synchronization data structure for objects stored locally;
- shares the responsibility of ensuring adequate replication with the other servers in the replica set; and
- makes decisions based only on differences detected between itself and its immediate neighbors.

Passing Tone uses *Merkle trees* for synchronization [4]. As a consequence of keeping only a single Merkle tree per server, reflecting that server's actual objects, a Passing Tone server must create a replica of objects that it is missing in order to avoid repeatedly learning about that object. The count of available replicas is maintained implicitly as servers synchronize with their neighbors: when the first k servers in an object's successor list have a replica, there are at least k replicas.

Like Carbonite [5], Passing Tone uses extra replicas to mask transient failures and provide durability. Each server in Passing Tone has two maintenance responsibilities. First, it must ensure that it has replicas of objects

```
n.local_maintenance():
    a, b = n.pred_k, n # k-th predecessor to n
    for partner in n.succ, n.pred:
        diffs = partner.synchronize(a, b)
        for o in diffs:
            data = partner.fetch(o)
            n.db.insert(o, data)
```

Figure 2: The local maintenance procedure ensures that each server n has replicas of objects for which it is responsible. The `synchronize(a, b)` method compares the local database with the partner's database to identify objects with keys in the range (a, b) that are not stored locally. Local maintenance does not delete objects locally or remotely.

for which it is responsible. This is its *local maintenance* responsibility. Second, it must ensure that objects it is no longer responsible for but has stored locally are offered to the new responsible server. This represents a server's *global maintenance* responsibility.

4.3 Local maintenance

The local maintenance algorithm distributes the responsibility of ensuring that sufficient replicas exist to each of the servers in the current replica set. To do this without requiring coordination across all servers simultaneously, Passing Tone's local maintenance relies on an extension to Chord that allows each server to know precisely which replica sets it belongs to: Passing Tone asks Chord to maintain a *predecessor list* which tracks the first $O(\log n)$ predecessors of each server. The details of this are discussed in Section 5. Once the predecessor list is available, each server will know the range of keys it is responsible for replicating and can identify whether or not it needs to replicate an object simply by considering the key relative to its first k predecessors.

The algorithm that implements local maintenance is shown in Figure 2. Each server synchronizes with only its direct predecessor and successor, over the range of keys for which it should be holding replicas, as determined by its Chord identifier. Synchronization walks down the relevant branches of the Merkle tree to efficiently identify objects that the server is missing but that are present on its neighbor.

Merkle trees store the keys of a server in a tree with a 64-way branching factor: each node of the tree stores a hash of the concatenation of the hashes of its children. At the bottom, the leaves of the tree represent the hash of the keys of the objects themselves. When two sub-trees represent exactly the same set of keys, the hashes of the root of the sub-trees will match: the synchronization protocol can detect this efficiently and avoid further process-

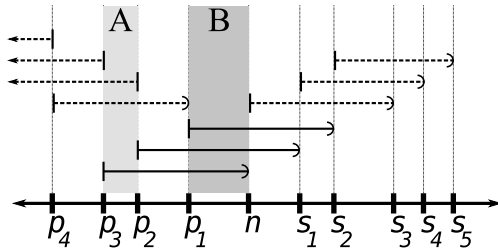


Figure 3: Server n is responsible for objects whose keys fall between its predecessor p_1 and itself. In the figure, objects are replicated with an initial replication level $k = 3$. Thus, objects that n is responsible for are replicated on s_1 and s_2 . The ranges of the objects held by each server is shown with horizontal intervals; the intersection with vertical regions such as A and B show which servers hold particular objects.

ing of those sub-trees. The synchronization protocol also restricts the branches considered based on the range to be synchronized over. This allows the same tree to be used to synchronize with any server.

Any objects that synchronization identifies as missing locally are then replicated locally. By asking each server to be responsible for replicating objects to itself, Passing Tone ensures that no single server need count the number of replicas of an object. Rather, the servers in a successor list operate independently but cooperatively to ensure the right number of replicas exist.

This approach ensures that the implementation will never need to maintain or consult information from multiple servers simultaneously. This reduces the memory and disk impact of maintenance. This also avoids the problem that can lead to spurious repairs: instead of accumulating information and referring to it after it has become stale, each server in Passing Tone make decisions immediately upon synchronizing with its neighbor.

Synchronizing with the successor and predecessor is sufficient to eventually ensure k replicas of any failed objects. This follows directly from how objects are arranged in a Chord consistent hashing scheme. Replicating objects from the successor will allow servers to recover from missed insertions or disk failures. Similarly, replicating objects from the predecessor will help servers cover for other servers that may have failed transiently.

This can be seen more clearly with reference to Figure 3. There are two cases to consider that cause object movement: the failure of a server, or the addition of a server. When server n fails, server s_1 becomes responsible for holding replicas of objects in the region labeled A. It can retrieve these replicas from its new predecessor p_1 ;

```
n.global_maintenance():
    a, b = n.pred_k, n # k-th predecessor to n
    key = n.db.first_succ(b) # first key after b
    while not between(a, b, key):
        s = n.find_successor(key)
        diffs = s.reverse_sync(key, s)
        for o in diffs:
            data = n.db.read(o)
            s.store(o, data)
        key = n.db.first_succ(s)
```

Figure 4: Global maintenance ensures objects are placed in the correct replica sets. n periodically iterates over its database, finds the appropriate successor for objects it is not responsible for and offers them to that successor. The *reverse_sync* call identifies objects present on n but missing on s over the specified range.

it is unlikely to retrieve such objects from its successor s_2 , though this is possible.

When n is joining the system, it divides the range of keys that its successor is responsible for: n is now responsible for keys in $[p_1, n)$ whereas s_1 is now responsible for $[n, s_1)$. In this case, n can obtain objects in region B from s_1 . If n was returning from a temporary failure, this will include objects inserted when n was absent.

Over time, even if an object is not initially present on the successor or predecessor, it will migrate to those servers because they themselves are executing the same maintenance protocol. Thus, as long as one server in a replica set has a replica of an object, it will eventually be propagated to all such servers.

Despite only local knowledge, local maintenance does not result in excessive creation of replicas. Temporary failures do not cause replicas to be repeatedly created and then deleted because Passing Tone allows objects to remain on servers even if there are already k replicas; objects are created once and “re-used” on subsequent failures. Our first maintenance algorithm [6] kept replicas on *exactly* the first k successors, which was shown to be a bad decision [5].

4.4 Global maintenance

While local maintenance focuses on objects that a server is responsible for but does not have, global maintenance focuses on objects that a server has but for which it is not responsible. Global maintenance ensures that, even after network partitions or other rapid changes in (perceived) system size, a server’s objects are located in a way that DHash’s read algorithm and Passing Tone’s local maintenance algorithm will be able to access them.

Global and local maintenance in Passing Tone are cooperative and complementary: global maintenance explicitly excludes those objects that are handled by local mainte-

nance. At the same time, global maintenance in Passing Tone relies on local maintenance to correctly propagate replicas to other servers in the official replica set. That is, once an object has been moved to the successor, the neighbors of the successor will take responsibility for creating replicas of that object across the replica set.

Figure 4 shows the pseudo-code for global maintenance. Like local maintenance, global maintenance requires the predecessor list to determine which objects each server should maintain. Server n periodically iterates over its database, identifying objects for which it is not in the replica set. For each such object, it looks up the current successor s and synchronizes with s over the range of objects whose keys fall in s 's range of responsibility. s then makes a copy of any objects that it is responsible for which it does not have a replica. Once these transfers are complete, global maintenance moves on to consider the next set of objects.

Like in local maintenance, the global maintenance algorithm does not delete replicas from the local disk, even if they are misplaced. These replicas serve as extra insurance against failures. Because the synchronization algorithm efficiently identifies only differences, once the objects have been transferred to the actual successor, future checks will be inexpensive.

4.5 Supporting expiration

The expiration time of an object merely acts as a guideline for deletion: a server can hold on to objects past their expiration if there is space to do so. As described above, however, the local and global maintenance algorithms do not take expiration into account.

Failure to account for expiration in maintenance can lead to two problems. First, repairing these objects are a waste of system resources, since expired objects are ones that the application has specified as no longer requiring durability. Second, when expired objects are not expired simultaneously across multiple servers, spurious repairs can occur if one server chooses to delete an expired object before its neighbor: the next local maintenance round could identify and re-replicate that object.

One solution to this problem would be to include extra metadata during synchronization so that repairs can be prioritized based on expiration time. Including the metadata initially seems attractive: especially when interacting with mutable objects, metadata may be useful for propagating updates in addition to allowing expiration-prioritized updates. OpenDHT implements a simple variant of this by directly encoding the insertion time into the Merkle tree [27]; this complicates the construction of Merkle trees however and requires a custom Merkle tree be constructed per neighbor.

To address these issues, Passing Tone separates object storage from the synchronization trees. The Merkle syn-

Process	kLoC	Role
usenetdht	3.4	Usenet front-end and DHash interface.
lsd	28.4	Chord routing, DHash gateway and server logic.
adbd	1.5	Main data storage.
maintd	6.8	Maintenance including Passing Tone and Merkle synchronization.

Table 1: UsenetDHT and DHT processes breakdown.

chronization tree contains only those keys stored locally that are not expired. Keys are inserted into the tree during inserts and removed when they expire. Expired objects are removed by DHash when the disk approaches capacity.

This expiration scheme assumes that servers have synchronized clocks—without synchronized clocks, servers with slow clocks will repair objects that other servers have already expired from their Merkle trees.

5 Implementation

This section provides an overview of our DHash, Passing Tone and UsenetDHT implementations. We highlight the particular problems and lessons that we learned while implementing these systems to meet our performance requirements.

5.1 Source code

DHash, Passing Tone and UsenetDHT are implemented in C++ using the `libasync` libraries [21]. Each host runs four separate processes, detailed in Table 1.

`usenetdht` implements NNTP to accept feeds and interact with Usenet clients. It stores each article as a single DHash content-hash object; use of a single object minimizes the overhead incurred by DHash during storage, relative to chunking objects into, for example, 8 Kbyte blocks. `usenetdht` maintains a database containing the overview metadata and content-hash for each article, organized by newsgroup.

For performance, the UsenetDHT implementation draws largely on techniques used in existing open-source Usenet servers. For example, UsenetDHT stores overview data in a BerkeleyDB database, similar to INN's overview database (`ovdb`). An in-memory history cache is used to remember what articles have been recently received. This ensures that checks for duplicate articles do not need to go to disk. The current UsenetDHT implementation does not yet support a DHT read-cache.

The DHash and Passing Tone implementation is structured into three processes to avoid having disk I/O calls block and delay responses to unrelated network requests. Network I/O and the main logic of Chord and DHash are handled by `lsd`. Disk I/O is delegated to `adbd`, which

handles storage, and `maintd`, which implements Passing Tone and handles maintenance.

The complete DHash source base consists of approximately 38,000 lines of C++. This includes the Chord routing layer and research code for previous DHT and maintenance algorithms. The Merkle synchronization subsystem consists of approximately 3,000 lines of code; the Passing Tone implementation and supporting infrastructure is less than 1,000 lines of code.

5.2 Predecessor lists

In performing maintenance, Passing Tone requires information about the objects for which each individual server is responsible. This requires that each server know its first *k* predecessors. Predecessor lists are maintained using the inverse of the Chord successor list algorithm. Each server periodically asks its current predecessor for their predecessor list. When providing a predecessor list to a successor, a server takes its own predecessor list, pops off the furthest entry and inserts itself.

Predecessor lists are not guaranteed to be correct. Servers are not considered fully joined into the Chord ring until they have received a notification from their predecessor. The predecessor maintenance algorithm may return a shortened, wrong or empty predecessor list in this situation. For this reason, this algorithm was rejected in the design of Koorde [18], which relies on predecessor lists for correct routing. However, this situation occurs fairly rarely in practice, if at all, and we have never observed any problems resulting from this implementation. Because Passing Tone does not rely on predecessor lists for routing, we accept the occasional possibility of error. Even in the event that an incorrect predecessor list is used for determining maintenance, Chord stabilization will soon cause the predecessor list to change and any outstanding incorrect repairs will be flushed before significant bandwidth has been used.

5.3 Load balance

DHash uses virtual servers for load balance [6]. To ensure that replicas are placed on virtual servers of different physical servers, `lsd` filters co-located virtual servers from the successor list during write operations, similar to Y0 [13]. Co-located virtual servers are similarly filtered from successor (and predecessor) lists during read and maintenance. However, currently, administrators must manually balance the number of virtual servers used so that each physical server does not receive more load than its network link or disk can support. We know of no algorithm that balances load over multiple constraints.

5.4 Expiration support

To support expiration in UsenetDHT, DHash now allows applications to provide a per-object expiration time.

DHash treats this time as a recommendation: DHash servers stop performing maintenance of expired objects but only delete the objects when storage is needed for new writes. In this way, the application maintains control of how long data is durably maintained, and DHash servers can operate without consulting the application in prioritizing objects for maintenance. The implementation of expiration in DHash is inspired by the timehash system used by INN (described in [31]).

5.5 Storage performance

`adbd` stores objects and metadata separately. It stores objects in append-only flat files, and stores metadata, including the Merkle synchronization tree, using BerkeleyDB databases. BerkeleyDB databases provide a simple key-value store, here used to map object keys to metadata. Storing objects outside of BerkeleyDB is important for performance. Since BerkeleyDB stores data and key within the same BTree page and since pages are fixed size, objects larger than a single page are stored on separate overflow pages. This results in particularly poor disk performance as overflow pages cause writes to be chunked into page-sized fragments that are not necessarily located contiguously, leading to additional disk seeks.

`adbd` names object storage files by approximate expiration time. Each file holds 256 seconds worth of objects, which works well for a system with high write workload. Files can be appended to efficiently, and object data can be read without extra seeks. Grouping multiple objects into a single file allows efficient reclamation of expired data when the disk approaches capacity: 256 seconds worth of objects can be freed by unlinking a single file.

`adbd` uses one database to map object keys to metadata such as the object's size, expiration time, and offset within the flat file storing the object data. A separate pair of databases stores information corresponding to the Merkle tree. `adbd` handles write requests and updates the Merkle tree as new objects are written and old ones are expired. `maintd` reads from this same tree during synchronization. BerkeleyDB's transaction system is used to provide atomicity, consistency and isolation between databases and between processes. For performance, the implementation disables writing the transaction log synchronously to disk.

Persistently storing the Merkle tree improves start-up performance. An earlier implementation stored object data with metadata and reconstructed an in-memory Merkle tree during start-up. BerkeleyDB could not efficiently enumerate the keys of the database; each page read returned few keys and much data. Further, BerkeleyDB does not include an API for enumerating the database in on-disk order. A straightforward reconstruction of the Merkle tree would require one seek per key over the entire set of keys held in the tree. With a 120 Gbyte disk (hold-

ing 600,000 173 Kbyte articles), at 5ms per seek, enumerating these keys could take over 50 minutes.

The Merkle tree implementation currently stores the pre-computed internal nodes of the Merkle tree in a separate database addressed directly by their prefix. Prefix addressing allows `maintd` to directly retrieve internal nodes during synchronization without traversing down from the root. `maintd` uses a BerkeleyDB internal in-memory cache, sized to keep most of the nodes of the Merkle tree in memory, retaining the performance benefits of an in-memory implementation. The implementation stores object keys in a separate database. Since keys are small, the backing database pages, indexed by expiration time, are able to hold dozens of keys per page, allowing key exchanges in Merkle synchronization to be done with few disk seeks as well.

6 Evaluation

In this section, we demonstrate the following:

- In a simulated PlanetLab environment, Passing Tone provides object durability;
- Our test deployment of UsenetDHT is able to support a live Usenet feed;
- Passing Tone identifies repairs following transient failures, permanent failures and server additions without interfering with normal operations; and
- Our implementation of DHash and UsenetDHT scales with the available resources.

6.1 Evaluation method

We evaluate the ability of Passing Tone to provide durability by considering its behavior using a trace-driven simulator. A trace of the disk and transient failures, from March 2005 through March 2006, on the PlanetLab test-bed [25] drives the simulation. The trace and simulator first appeared in prior work [5].

Passing Tone is evaluated under load using a live wide-area deployment. The deployment consists of twelve machines at universities in the United States: four machines are located at MIT, two at NYU, and one each at the University of Massachusetts (Amherst), the University of Michigan, Carnegie Mellon University, the University of California (San Diego), and the University of Washington. Access to these machines was provided by colleagues at these institutions and by the RON test-bed. While these servers are deployed in the wide area, they are relatively well connected, many of them via Internet2.

Unlike PlanetLab hosts, these machines are lightly loaded, very stable and have high disk capacity. Each machine participating in the deployment has at least a single 2.4 Ghz CPU, 1 Gbyte of RAM and UDMA133 SATA disks with at least 120 Gbyte free. These machines are not directly under our control and are shared with other users; write caching is enabled on these machines.

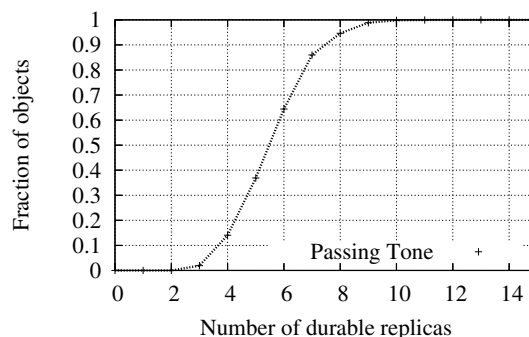


Figure 5: Durability of objects in Passing Tone: the graph shows a CDF of the number of replicas per object after a one year PlanetLab trace. No objects are lost during this simulation.

The load for the live deployment is a mirror of the CSAIL news feed; the CSAIL feed is ranked 619th in the Usenet Top 1000 servers [11]. This feed sees one million articles per day on average and is around 10% of the full feed. This feed generates a roughly continuous 2.5 Mbyte/s of write traffic. Binary articles have a median article size of 240 Kbyte; text articles are two orders of magnitude smaller, with a median size of 2.4 Kbyte.

Microbenchmarks, run on a private gigabit switched local area network, demonstrate the scalability potential of DHash and UsenetDHT. Using a local network eliminates network bottlenecks and focuses on the disk and memory performance.

6.2 Passing Tone durability

We evaluate Passing Tone's durability in simulation over a real-world PlanetLab trace. There are a total of 632 unique hosts experiencing 21,255 transient failures and 219 disk failures. Failures are not evenly distributed, with 466 hosts experiencing no disk failures, and 56 hosts experiencing no disk or transient failures. At the start of the trace, 50,000 20 Mbyte objects are inserted and replicated according to standard DHash placement. With an average of 490 online servers and $k = 2$ replicas, this corresponds to just over 4 Gbyte of data per server. To approximate PlanetLab bandwidth limits, each server has 150 Kbyte/s of bandwidth for object repairs. At this rate, re-creating the contents of a single server takes approximately 8 hours.

For Passing Tone to be viable, it must not lose any objects. Passing Tone synchronizes with either its predecessor or successor to generate repairs every ten minutes or when these servers change. Figure 5 shows a CDF of the number of replicas for each object at the end of the trace: the most important feature is that all objects have at least three replicas. No objects are lost, showing that Passing

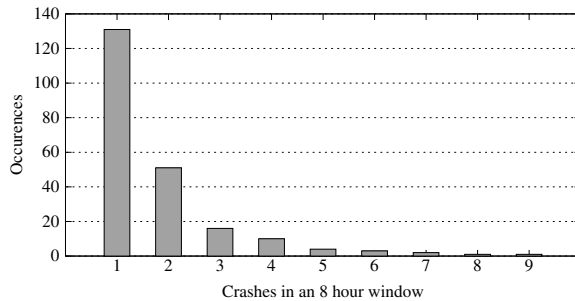


Figure 6: Number of crashes within an eight hour window over the course of the PlanetLab trace. Eight hours represents the approximate time needed to replicate the data of one PlanetLab server to another.

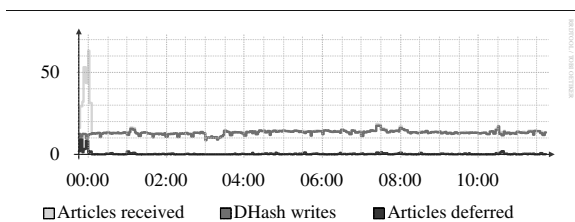


Figure 7: Articles received, posted and deferred by the CSAIL UsenetDHT deployment over a twelve hour period. During normal operation, deferrals are very rare.

Tone can provide durability in a PlanetLab environment.

The CDF also demonstrates that some objects have many more replicas; ten percent have over seven replicas. To understand why, consider Figure 6, which shows the number of times a given number of servers crashed (i.e., lost disks) within an 8 hour period over the entire trace; the worst case data loss failure in the trace could only be protected if at least nine replicas existed for objects on the nine simultaneously failing servers. It is unlikely that the actual failure in the trace would have mapped to a contiguous set of servers on a Chord ring; however, the simulation shows that Passing Tone saw sufficient transient failures to create nine replicas in at least some cases.

6.3 UsenetDHT wide-area performance

The CSAIL Usenet feed receives on average 14 articles per second. This section demonstrates that UsenetDHT, with DHash and Passing Tone, is able to meet the goal of supporting CSAIL's Usenet feed. DHash is configured to replicate articles twice, prior to any maintenance; thus to support the CSAIL feed, the system must write at least 28 replicas per second. Our deployment has supported this feed since July 2007. This section also demonstrates that our implementation supports distributed reads at an aggregate 30.6 Mbyte/s.

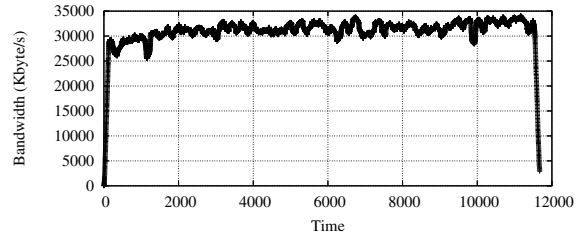


Figure 8: UsenetDHT aggregate read throughput. 100 hosts started article reads continuously against 12 servers.

Figure 7 shows the number of articles received by UsenetDHT, posted into DHash writes, and deferred. Deferrals occur when DHash's write window is full—during initial server startup, deferrals occur since there is a backlog of posts from the upstream feed that arrive faster than the local DHash server can write replicas to remote sites. The upstream later re-sends deferred posts. During normal operation, deferrals do not occur, showing that UsenetDHT and DHash can keep up with the normal workload from the CSAIL feed.

To evaluate the read capability of the system, we use clients distributed on the PlanetLab test bed. We selected 100 of the servers with the lowest load that were geographically close to one of the seven sites that were running at the time of the experiment using CoMon [24]. Each client machine ran five parallel NNTP clients to the UsenetDHT front-end closest to them and downloaded articles continuously from newsgroups chosen at random.

Figure 8 plots the achieved aggregate bandwidth over time: the client machines were able to achieve an collectively 30.6 Mbyte/s or approximately 313 Kbyte/s per client machine. This corresponds to reading on median 2612 Kbyte/s per disk. We estimate that each article read requires seven disk seeks including metadata lookup at the Usenet server, object offset lookup, opening the relevant file and its inode, indirect and double-indirect blocks, data read, and atime update. With at least 70ms of total seek time per access, this means that each disk can support no more than 14 reads per second: given an average article size of 173 Kbyte ($14 \times 173 = 2422$ Kbyte/s), this corresponds well with the observed throughput per disk.

6.4 Passing Tone efficiency

When failures do arise, Passing Tone is able to identify repairs without substantially affecting the ability of the system to process writes. Figure 9 demonstrates the behavior of Passing Tone under transient failures and server addition in a six hour window, by showing the total number of objects written: the number of objects repaired is shown in a separate color over the number of objects written. At 18:35, we simulated a 15 minute crash-and-reboot

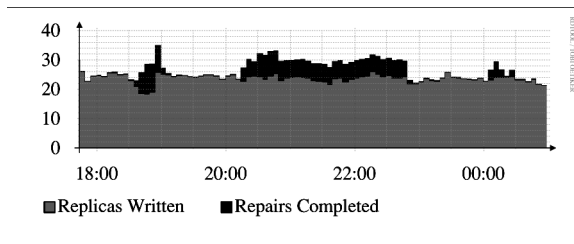


Figure 9: Number of repairs and number of writes over time, where we introduce a simple transient failure, a server addition, and a second transient failure. The second failure demonstrates behavior under recovery and also what would have happened in a permanent failure.

cycle on a server responsible for 8.5% of the data in the system. Passing Tone immediately begins repairing and ceases when that server comes back online at 18:50. A more permanent failure would involve a longer period of repairs, as repair speed is limited largely by bandwidth.

To demonstrate this limit, the next event occurs at 20:15, where we add a new server, responsible for 3% of the data in the system. It begins transferring objects from its successor and moves 11 Gbyte of data in just over 2.4 hours. This corresponds to 1.2 Mbyte/s, which is what the link between that server and its successor can support.

Finally, we demonstrate a short transient failure of the same new server at 00:05. Its neighbors begin repairing the objects that were inserted to the failed server during its four hour lifetime: however, none of the objects that were stored on the original successor needed to be repaired because its replicas were not deleted. When the new server is brought back online at 00:20, there are a few transient repairs to bring it up to date with the objects that were written during its downtime but not a repeat of the four hour initial transfer. This second transient failure also indicates how Passing Tone would behave after a permanent failure: we expect the transient failures will create a small buffer of extra replicas so that when a permanent failure does occur, only those (relatively) few objects that have been inserted since will require re-replication.

6.5 DHash microbenchmarks

The microbenchmarks were conducted on a cluster of 8 Dell PowerEdge SC1425 servers, with Intel® Xeon™ 2.80 Ghz CPUs (HyperThreading disabled), 1 Gbyte of RAM, and dual Maxtor SATA disks. Each machine runs FreeBSD 5.4p22. The machines are each directly interconnected with a Gigabit Ethernet switch. These machines are under our direct control and write caching is disabled. They can do sustained writes at 7 Mbyte/s on average and read at 40 Mbyte/s; a half stroke seek takes approximately 14ms. A separate set of machines are used for generating client load.

Number of DHT servers	Median bandwidth (Kbyte/s)	
	Write	Read
1	6600	14000
2	12400	19000
3	18800	28000
4	24400	33900
5	30600	42600
6	36200	49200
7	42600	57200
8	46200	63300

Table 2: Read and write microbenchmark performance.

To test the read and write scalability of the implementation, we configure one client per DHT server and direct each client to write (and read) a 2 Gbyte synthetic stream of 200 Kbyte objects as fast as possible. Replication, maintenance and expiration are disabled. The resulting load is spread out over all the servers in the system. The results are shown in Table 2.

Each individual machine contributes on average an additional 5.7 Mbyte/s of write throughput; thus in an environment that is not network constrained, our implementation easily operates each server at close to its disk bottleneck. For reads, each additional machine contributes on average 7 Mbyte/s of read throughput. This result is largely driven by seeks. In the worst case, each 200 Kbyte object requires at least one seek to look up the object's offset in the metadata database, one seek to read the object and possibly one seek to update the atime on the inode. In this case, it would require 40ms simply to seek, limiting the number of objects read per disk to 25 per second (or ≈ 5 Mbyte/s). Systems with fewer servers will do better on this particular benchmark as objects are read in order of insertion. Thus, with fewer write workloads intermingled, fewer seeks will be required and operating system read-ahead may work well. The observed 14 Mbyte/s in the single server case corresponds to one seek per read on average, where it is likely that BerkeleyDB has cached the offset pages and the OS read-ahead is successful. In the multi-server cases, the 7 Mbyte/s average increase corresponds well to two seeks per read.

Thus, in a well-provisioned local network, the DHash implementation can write a 2.5 Mbyte/s Usenet feed to a single server (though with extremely limited retention). By adding additional servers (and hence disk arms), DHash is also able to scale as needed to support reader traffic and increase retention.

7 Related work

The problems of costs from repetitive article transmission and replication of un-read articles in Usenet have been noted previously. Newscaster [2] examined using IP mul-

ticast to transfer news articles to many Usenet servers at once. Each news article only has to travel over backbone links once, as long as no retransmissions are needed. In addition, news propagation times are reduced. However, Newscaster still requires that each Usenet server maintain its own local replica of all the newsgroups. IP multicast also requires extensive infrastructure within the network, which is not required by overlays such as DHash.

NewsCache [15] is one of several projects that reduce bandwidth at servers by caching news articles. It is designed to replace traditional Usenet servers that are leaf nodes in the news feed graph, allowing them to only retrieve and store articles that are requested by readers. In addition to filling the cache on demand, it can also pre-fetch articles for certain newsgroups. These features are also available as a mode of DNews [23], a commercial high-performance server, which adds the ability to dynamically determine the groups to pre-fetch. Both NewsCache and DNews reduce local bandwidth requirements to be proportional to readership and use local storage for caching articles of interest to local readers. UsenetDHT employs these caching strategies but also makes use of a DHT to remove the need to pay an upstream provider to source the articles.

The Coral and CobWeb CDNs provide high-bandwidth content distribution using DHT ideas [9, 37]. Coral uses distributed sloppy hash tables that optimize for locality and CobWeb locates data using Pastry [3] (with content optimally replicated using Beehive [26]). However, CDNs cache content with the aim of reducing latency and absorbing load from an origin server. UsenetDHT operates in a system without origin servers and must guarantee durability of objects stored.

Dynamo is an Amazon.com-internal DHT and uses techniques for load balance, performance and flexibility that are similar to those used in DHash and Passing Tone [8]. Dynamo is deployed in cluster-oriented environments, with fewer wide area links than DHash. OpenDHT is deployed on PlanetLab and provides a DHT for public research use [27, 28]. Compared to DHash, OpenDHT focuses on storing small data objects. Like DHash, OpenDHT provides time-to-live (TTLs) but uses them to guarantee fair access to storage across multiple applications. Both Dynamo and OpenDHT also use Merkle trees for synchronization but without explicit support for expiration.

Passing Tone optimizes maintenance for accuracy and dealing with expiration. Like Carbonite, Passing Tone keeps extra copies of objects on servers in the successor list to act as a buffer that protects the number of available objects from falling during transient failures [5]. Like Passing Tone, PAST allows more than k replicas of an object but PAST's replicas are primarily for performance not durability [30]; Passing Tone manages disk capacity with

expiration, not diversion. Passing Tone's division of local and global maintenance originally was proposed by Cates [4] and has also been used in OpenDHT [27].

Despite a high number of objects, Passing Tone's use of Merkle synchronization trees [4] reduces bandwidth use during synchronization, without the complexity of aggregation as used in Glacier [16]. The idea of eventual consistency in Passing Tone is similar to Glacier's use of rotating Bloom filters for anti-entropy. However, Bloom filters still repetitively exchange information about objects and, at some scales, may be infeasible. Minsky *et al*'s synchronization algorithm [22] is more network efficient than Merkle trees but available implementations do not support persistence; in practice synchronization bandwidth is dwarfed by data transfer bandwidth.

8 Conclusions

After three decades, Usenet continues to be an important network service because of its distinct advantages over other data distribution systems. This results in over 1 Tbyte of new content posted to Usenet per day. Usenet servers have improved dramatically to carry this level of load, but the basic Usenet design hasn't changed, even though its flooding approach to distributing content is expensive. With the current design only a limited of servers can provide the full Usenet feed. We propose to exploit the recent advances in DHTs to reduce the costs of supporting Usenet, using a design that we call UsenetDHT.

UsenetDHT aggregates n servers into a DHT that stores the content of Usenet. This approach reduces the costs of storing and receiving a feed to $O(1/n)$. To enable a DHT to store as much data as Usenet generates, we developed the Passing Tone maintenance algorithm, which provides good durability and availability, keeps the memory pressure on servers low and avoids disks seeks. Experiments with a small deployment show that Passing Tone and UsenetDHT support the 2.5 Mbyte/s feed at CSAIL and should be able to scale up to the full feed by adding more servers. These results suggest that UsenetDHT may be a promising approach to evolve Usenet and to allow it to continue to grow.

Acknowledgments Frank Dabek and James Robertson are co-authors for an earlier portion of this work published at IPTPS 2004 [32]. Garrett Wollman provided the live Usenet feed used for evaluation and graciously answered many questions. The wide-area results made use of the PlanetLab and the RON test-beds; also, Magda Balazinska, Kevin Fu, Jinyang Li, and Alex Snoeren made resources at their respective universities available to us for testing. Finally, this paper improved considerably due to the comments and suggestions of the reviewers and our shepherd, Emin Gün Sirer.

References

- [1] BARBER, S. Common NNTP extensions. RFC 2980, Network Working Group, Oct. 2000.
- [2] BORMANN, C. The Newscaster experiment: Distributing Usenet news via many-to-more multicast. <http://citeseer.nj.nec.com/251970.html>.
- [3] CASTRO, M., DRUSCHEL, P., HU, Y. C., AND ROWSTRON, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research, June 2002.
- [4] CATES, J. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.
- [5] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006).
- [6] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001).
- [7] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).
- [8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating System Principles* (Oct. 2007).
- [9] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with Coral. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).
- [10] GHANE, S. Diablo statistics for spool-1t2.cs.clubint.net (TOP1000 #24). <http://usenet.clubint.net/spool-1t2/stats/>. Accessed 30 September 2007.
- [11] GHANE, S. The official TOP1000 Usenet servers page. <http://www.top1000.org/>. Accessed 13 September 2007.
- [12] GIGANEWS. 1 billion Usenet articles. <http://www.giganews.com/blog/2007/04/1-billion-usenet-articles.html>, Apr. 2007.
- [13] GODFREY, P. B., AND STOICA, I. Heterogeneity and load balance in distributed hash tables. In *Proc. of the 24th Conference of the IEEE Communications Society (Infocom)* (Mar. 2005).
- [14] GRADWELL.COM. Diablo statistics for news-peer.gradwell.net. <http://news-peer.gradwell.net/>. Accessed 30 September 2007.
- [15] GSCHWIND, T., AND HAUSWIRTH, M. NewsCache: A high-performance cache implementation for Usenet news. In *Proc. of the 1999 USENIX Annual Technical Conference* (June 1999), pp. 213–224.
- [16] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).
- [17] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND JAMES W. O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation* (Oct. 2000).
- [18] KAASHOEK, F., AND KARGER, D. Koorde: A simple degree-optimal distributed hash table. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems* (Feb. 2003).
- [19] KANTOR, B., AND LAPSLEY, P. Network news transfer protocol. RFC 977, Network Working Group, Feb. 1986.
- [20] KARGER, D., AND RUHL, M. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004).
- [21] MAZIÈRES, D. A toolkit for user-level file systems. In *Proc. of the 2001 USENIX Annual Technical Conference* (June 2001).
- [22] MINSKY, Y., TRACHTENBERG, A., AND ZIPPEL, R. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory* 49, 9 (Sept. 2003), 2213–2218. Originally published as Cornell Technical Report 2000-1796.
- [23] NETWIN. DNews: Unix/Windows Usenet news server software. <http://netwinsite.com/dnews.htm>. Accessed 9 November 2003.
- [24] PARK, K. S., AND PAI, V. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review* 40, 1 (Jan. 2006), 65–74. <http://comon.cs.princeton.edu/>.
- [25] PETERSON, L., BAVIER, A., FIUCZYNSKI, M. E., AND MUIR, S. Experiences building PlanetLab. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, Nov. 2006).
- [26] RAMASUBRAMANIAN, V., AND SIRER, E. G. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).
- [27] RHEA, S. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, 2005.
- [28] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: A public DHT service and its uses. In *Proc. of the 2005 ACM SIGCOMM* (Aug. 2005).
- [29] ROLFE, A. Private communication, 2007.
- [30] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001).
- [31] SAITO, Y., MOGUL, J. C., AND VERGHESE, B. A Usenet performance study. <http://www.research.digital.com/wrl/projects/newsbench/usenet.ps>, Nov. 1998.
- [32] SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead Usenet server. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004).
- [33] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160.
- [34] SWARTZ, K. L. Forecasting disk resource requirements for a Usenet server. In *Proc. of the 7th USENIX Large Installation System Administration Conference* (1993).
- [35] Network status page. <http://www.usenetserver.com/en/networkstatus.php>. Accessed 30 September 2007.
- [36] WOLLMAN, G. Private communication, 2007.
- [37] YEE JIUN SONG AND, V. R., AND SIRER, E. G. Optimal resource utilization in content distribution networks. Computing and Information Science Technical Report TR2005-2004, Cornell University, Nov. 2005.

San Fermín: Aggregating Large Data Sets using a Binomial Swap Forest[†]

Justin Cappos and John H. Hartman
Department of Computer Science, University of Arizona

Abstract

San Fermín is a system for aggregating large amounts of data from the nodes of large-scale distributed systems. Each San Fermín node individually computes the aggregated result by swapping data with other nodes to dynamically create its own binomial tree. Nodes that fall behind abort their trees, thereby reducing overhead. Having each node create its own binomial tree makes San Fermín highly resilient to failures and ensures that the internal nodes of the tree have high capacity, thereby reducing completion time.

Compared to existing solutions, San Fermín handles large aggregations better, has higher completeness when nodes fail, computes the result faster, and has better scalability. We analyze the completion time, completeness, and overhead of San Fermín versus existing solutions using analytical models, simulation, and experimentation with a prototype built on peer-to-peer system deployed on PlanetLab. Our evaluation shows that San Fermín is scalable both in the number of nodes and in the aggregated data size. San Fermín aggregates large amounts of data significantly faster than existing solutions: compared to SDIMS, an existing aggregation system, San Fermín computes a 1MB result from 100 PlanetLab nodes in 61–76% of the time and from 2-6 times as many nodes. Even if 10% of the nodes fail during aggregation, San Fermín still includes the data from 97% of the nodes in the result and does so faster than the underlying peer-to-peer system recovers from failures.

1 Introduction

San Fermín aggregates large amounts of data from distributed nodes quickly and accurately. As distributed systems become more prevalent this is an increasingly important operation: for example, CERT logs about 1/4 TB of data daily on approximately 100 nodes distributed throughout the Internet [9]. Analysts use these logs to detect anomalous behavior that signals worms and other attacks, and must do so quickly to minimize damage. An example query might request the number of flows to and from each TCP/UDP port (to detect an anomalous distribution of traffic indicating an attack). In this example there are many flow counters per node and the requester is interested in the sum of each counter across all nodes. It is important that the data be aggregated quickly, as time is of the essence when responding to attacks, and accurately, as the aggregated result should include data from

as many nodes as possible and the data from each node exactly once. The more accurate the result, the more useful it is.

In San Fermín the properties of current networks are leveraged to build an efficient content aggregation network for large data sizes. Since core bandwidth is typically not the bottleneck [12], San Fermín allows disjoint pairs of nodes to communicate simultaneously, as they will likely not compete for bandwidth. A San Fermín node also sends and receives data simultaneously, making efficient use of full-duplex links. The result is that San Fermín aggregates large data sets significantly faster than existing solutions, on average returning a 1 MB aggregation from 100 PlanetLab nodes in 61–76% the time and from approximately 2-6 times as many nodes as SDIMS, an existing aggregation system. San Fermín is highly failure resistant and with 10% node failures during aggregation still includes the data from over 97% of the nodes in the result — and in most cases does so faster than the underlying peer-to-peer system recovers from failures.

San Fermín uses a *binomial swap forest* to perform the aggregation, which is well-suited to tolerate failures and take advantage of the characteristics of the Internet. In a binomial swap forest each node creates its own binomial tree by repeatedly swapping aggregate data with other nodes. This makes San Fermín highly resilient to failures because a particular node's data is aggregated by an exponentially increasing number of nodes as the aggregation progresses. Similarly, the number of nodes included in a particular node's aggregate data also increases exponentially as the aggregation progresses. Each node creates its own binomial swap tree; as long as at least one node remains alive San Fermín will produce a (possibly incomplete) aggregation result.

Having each node create its own binomial swap tree is highly fault-tolerant and fast, but it can lead to excessive overhead. San Fermín reduces overhead by pruning small trees that fall behind larger trees during the aggregation, as the small trees are unlikely to compute the result first and therefore increase overhead without improving speed or accuracy. When a tree falls behind San Fermín prunes it — the name San Fermín is derived from this behavior, after the festival with the running of the bulls in Pampalona.

1.1 Applications

In addition to CERT, San Fermín also benefits other applications that aggregate large amounts of data from

[†]This work was supported in part by the NSF under grant CCR-0435292

many nodes:

Software Debugging Recent work on software debugging [19] leverages execution counts for individual instructions. This work shows that the total of all the instruction execution counts across multiple nodes helps the developer quickly identify bugs.

System Monitoring Administrators of distributed systems must process the logs of thousands of nodes around the world to troubleshoot difficulties, track intrusions, or monitor performance.

Distributed Databases A common query in relational databases is GROUP BY [25]. This query combines table rows containing the same attribute value using an aggregate operator (such as SUM). The query result contains one table row per unique attribute value. In distributed databases different nodes may store rows with the same attribute value. The values at these rows must be combined and returned to the requester.

These applications are similar because they aggregate large amounts of data from many nodes. For example, for the CERT example, finding the distribution of ports on UDP and TCP flows seen in the last hour takes 512 KB (assuming 4 byte counters). In the software debugging application, tracking a small application like `bc` requires 40KB of counters. Larger applications may require more than 1MB of counters. The target environments may contain hundreds or thousands of nodes, forcing the aggregation to tolerate failures.

The aggregation function has similar characteristics for these applications as well. The aggregation functions are commutative and associative but may be sensitive to duplication. Typically, the aggregate data from multiple nodes is approximately the same size as any individual node's data.

The aggregation functions may also be sensitive to partial data in the result. If, for example, the data from a node is split and aggregated separately using different trees, the root may receive only some of the node's data. For applications that want distributions of data (such as the target applications) it may be important to either have all of a node's data or none of it.

In some cases it may be possible to compress aggregate data before transmission to reduce space. Such techniques are complimentary to this work. Some environments may require administrative isolation. This work assumes that the aggregation occurs in a single administrative domain with cooperative nodes.

2 Binomial Swap Forest

A binomial swap forest is a novel technique for aggregating data in which each node individually computes the aggregate result by repeatedly swapping (exchanging) aggregate data with other nodes. Two nodes swap data by sending each other the data they have aggregated

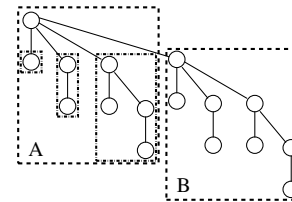


Figure 1: A 16-node binomial tree created by making tree B a child of tree A. The children of the root are themselves binomial trees of size 1, 2, 4, and 8.

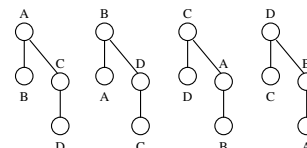


Figure 2: The binomial swap forest created by aggregating data from nodes A, B, C, and D. Each tree represents the sequence of swaps its root node performed while aggregating the data.

so far, allowing each to compute the aggregation of both nodes' data. The swaps are organized so that a node only swaps with one other node at a time, and each swap roughly doubles the number of nodes whose data is included in a node's aggregate data, so that the nodes will compute the aggregate result in roughly $\log(N)$ swaps. If the nodes of the aggregation are represented as nodes in a graph, and swaps as edges in the graph, the sequence of swaps performed by a particular node form a binomial tree with that node at the root. As a reminder, in a binomial tree with 2^n nodes the children of the root are themselves binomial trees with 2^{n-1} , 2^{n-2} , ..., 2^1 , and 2^0 nodes (Figure 1). As the figure illustrates, a binomial tree with 2^n nodes can be made from two binomial trees with 2^{n-1} nodes by making one tree a child of the other tree's root. The collection of binomial swap trees constructed by the nodes during a single aggregation is a *binomial swap forest*.

For example, consider data aggregation from four nodes: A, B, C, and D (Figure 2). Each node initially finds a partner with whom to swap data. Suppose A swaps with B and C swaps with D, so that afterwards A and B have the aggregate data AB, while C and D have the aggregate data CD. To complete the aggregation each node must swap data with a node from the other pair. If A swaps with C and B swaps with D, then every node will have the aggregate data ABCD.

The swaps must be carefully organized so that the series of swaps by a node produces the correct aggregated result. Consider aggregating data from $N = 2^n$ nodes each with a unique ID in the range $[0..N - 1]$ (we will later relax these constraints). Since each swap doubles the amount of aggregate data a node has, just prior to the last swap a node must have the data from half of the

Nodes	\hat{L}_2	\hat{L}_1	\hat{L}_0
000	Swap 001	Swap 010	Swap 101
001	Swap 000	Abort	
010	N/A	Swap 000	Swap 110
101	N/A	Swap 110	Swap 000
110	Swap 111	Swap 101	Swap 010
111	Swap 110	Abort	

Figure 3: One way 6 nodes can construct binomial swap forest. Each node swaps data with a node in each \hat{L}_k starting with \hat{L}_m and ending with \hat{L}_0 .

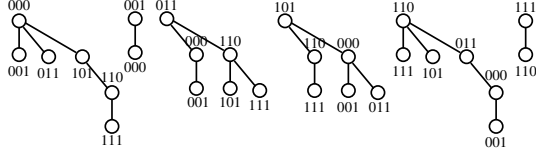


Figure 4: The binomial swap forest resulting from the construction in Figure 3. Nodes 001 and 111 aborted.

nodes in the system, and must swap with a node that has the data from the other half of the nodes. This can be achieved by swapping based on node IDs; specifically, if the node ID for a node x starts with a 0 then node x should aggregate data from all nodes that start with a 0 prior to the last swap, then swap with a node y whose node ID starts with 1 that has aggregated data from all nodes that start with a 1. Note that it doesn't matter *which* node y node x swaps with as long as its node ID starts with a 1 and it has successfully aggregated data from its half of the node ID space. Also note that node x should swap with exactly one node from the other half of the address space, otherwise the result may contain duplicate data. Recursing on this idea, assuming that node x starts with 00 then in the penultimate swap it must swap with a node whose node ID starts with 01 thus aggregating data from all nodes that start with 0. Similarly, in the very first swap node x swaps with the node whose node ID differs in only the least-significant bit. This is the general idea behind using a binomial swap forest to aggregate data — each node starts by swapping data with the node whose node ID differs in only the least-significant bit and works its way through the node ID space until it swaps with a node whose node ID differs in the most-significant bit.

Before describing this process in more detail it is useful to define the *longest common prefix*, \hat{L} of two nodes, which is the number of high-order bits the two node IDs have in common. We will use the notation $\hat{L}(x, y) = k$ to mean that the \hat{L} of nodes x and y is k bits long. With respect to a particular node x , we use the notation \hat{L}_k^x to indicate the set of nodes whose longest common prefix with node x is k bits long. We shorten this to \hat{L}_k when it is clear which node x is being referred to.

Using this notation, to aggregate data using a binomial

swap tree in a system with $N = 2^n$ nodes a node x must first swap data with a node in \hat{L}_{n-1}^x (there is only 1 node in this set), then swap data with a node in \hat{L}_{n-2}^x , etc., until eventually swapping data with a node in \hat{L}_0^x (there are 2^{n-1} nodes in this set). Again, node x swaps with only one node in \hat{L}_k to prevent duplication in the result. Each set \hat{L}_k^x has 2^{n-k-1} nodes, and node x will perform n swaps. Duplication cannot happen because when node x swaps data with node y from set \hat{L}_k^x , node x receives the data from nodes whose longest common prefix with node x is exactly k bits long. To see why this is true, consider that y has data from all nodes whose longest common prefix with y is at least $k+1$ bits. This means that the first k bits of these nodes are the same as y and since x differs with y in the k th bit, x must differ with these nodes in the k th bit.

The discussion so far assumes that the number of nodes in the system is a power of 2, that node IDs are in the range $[0..N-1]$, that each node knows how to contact every other node in the system directly, and that nodes do not fail. It also ignores the overhead of having each node construct its own binomial swap tree when only a single tree is necessary to compute the aggregated result. We can relax the first of these restrictions to allow the number of nodes to not be a power of 2, but it introduces several complications. First, the resulting binomial trees will not be complete, although they will produce the correct aggregate result. Consider data aggregation in a system with only nodes A, B, and C. Suppose A initially swaps with B. C must wait for A and B to finish swapping before it can swap with one of them. Suppose C subsequently swaps with A, so that both A and C have the aggregate data ABC, while node B only has AB. A and C successfully computed the result although the binomial trees they constructed are not complete. B was unable to construct a tree containing all the nodes.

Second, some nodes may not be able to find partners with whom to swap, as is the case with node B in the previous example. More generally, consider a collection of nodes whose longest common prefix \hat{L} is k bits long. To aggregate the data for that prefix the subset of nodes whose \hat{L}_{k+1} ends with a 0 must swap data with the subset whose \hat{L}_{k+1} ends with a 1. If these subsets are not of equal size, then some nodes will be unable to find a partner. Only if N is a power of 2 can the two subsets have equal numbers of nodes, otherwise some nodes will be unable to find a partner and must abort their aggregations.

Third, if the number of nodes is not a power of 2 then some node IDs will not be assigned to nodes. This can result in no nodes having a particular prefix, so that when other nodes try to swap with nodes having that prefix they cannot find a partner with whom to swap. Instead of aborting those nodes should instead simply skip the pre-

fix as it is empty. This is most likely to occur when the nodes initially start the aggregation process, as for any node x \hat{L}_n^x corresponds to exactly one node ID, which may not be assigned to a node. Therefore, instead of starting the aggregation with \hat{L}_n^x node x should instead initially swap with a node in \hat{L}_m^x where m is the longest prefix length for which \hat{L}_m^x is not empty.

As an example of aggregating data when N is not a power of 2, suppose that there are 6 nodes: 000, 001, 010, 101, 110, and 111 (Figures 3 and 4). Each node x swaps data with a node in each \hat{L}_k^x starting with \hat{L}_m^x and ending with \hat{L}_0^x . There are many valid binomial swap forests that could be constructed by these nodes aggregating data; in this example 000 first swaps with 001 and 110 swaps with 111. \hat{L}_2 is empty for 010 and 101, so they swap with nodes in \hat{L}_1 : 000 swaps with 010 and 101 swaps with 111. 001 and 110 cannot find a node in \hat{L}_1 with whom to swap (since 010 swapped with 000 and 101 swapped with 111) and they stop aggregating data. In the final step the remaining nodes swap with a node in \hat{L}_0 : 000 swaps with 101 and 010 swaps with 111.

The swap operations in a binomial swap forest are only partially ordered – the only constraints are that nodes must swap with a node in each \hat{L}_k in order starting with \hat{L}_m and ending with \hat{L}_0 . It is possible that in Figure 3 that nodes 000 and 010 will finish swapping before 111 and 110 finish swapping. This means that the only synchronization between nodes is when they swap data (there is no global synchronization between nodes).

San Fermín makes use of an underlying peer-to-peer communication system to handle both gaps in the node ID space and nodes that are not able to communicate directly. It uses time-outs to deal with node failures, and employs a pruning algorithm to reduce overhead by eliminating unprofitable trees. Section 4 these aspects of San Fermín in more detail.

3 Analytic Comparison

Several techniques have been proposed for content aggregation. The most straightforward is to have a single node retrieve all data and then aggregate. Some techniques like SDIMS [31] build a tree with high-degree nodes that are likely to have simultaneous connections. To provide resilience against failures, data is retransmitted when nodes fail. Seaweed [22] also has high-degree nodes with a similar structure to SDIMS, but uses a supernode approach in which the data on internal nodes are replicated to tolerate failures.

3.1 Analytic Models

Analytic models of these techniques enable comparison of their general characteristics. The models assume that any node that fails during the aggregation does not recover, and any node that comes online during the aggre-

	Description	Value	Source
N	Number of nodes	300,000	CorpNet [22]
b	Bandwidth	1.105Mbps	PlanetLab
l	Latency	190ms	AllSitesPing [2]
s	Data size	1MB	CERT [9]
c	Per node failure prob.	$5.5 * 10^{-6}$ / sec.	Farsite [22]
r	Supernode replicas	4	Seaweed [22]
d	Node degree	16	Seaweed [22]

Table 1: Model parameters.

gation does not join it. The probability of a given node failing in the next second is c . Node failures are assumed to be independent. A node that fails while sending data causes the partial data to be discarded. Inter-node latencies and bandwidths are a uniform l and b , respectively. The bandwidth b is per-node, which is consistent with the bandwidth bottleneck existing at the edges of the network and not in the middle. Each node contributes data of size s and the aggregation function produces aggregate data of size s . Per-packet, peer-to-peer, and connection establishment costs are ignored for all techniques.

Other parameters such as the amount of data aggregated, speed and capacity of the links, etc. are derived from real-world measurements (Table 1). The bandwidth measurements were gathered by transferring a 1MB file to all PlanetLab nodes from several well-connected nodes. The average bandwidth was within 100 Kbps for all runs, independent of the choice of source node. This means that well-connected nodes have roughly the same bandwidth to other nodes regardless of network location. The average of all runs is used in Table 1.

For each technique its completion time, completeness (number of nodes whose data is included in the aggregate result), and overhead are analyzed. Rather than isolating all of the parameters for each technique, the data size and number of nodes are varied to show their effect.

3.2 Binomial Swap Forest (San Fermín)

The analysis of San Fermín assumes a complete binomial swap forest. Since it takes $\frac{s}{b} + l$ time to do a swap, the completion time is $\log_2(N) * (\frac{s}{b} + l)$. Figures 5a and 6a show that using a binomial swap forest is effective at rapidly aggregating data. For example, using a binomial swap forest takes less than 1/3 the time of other techniques when more than 128 KB of data per node is aggregated.

After a node swaps with n other nodes in a binomial swap forest its data will appear in 2^n binomial trees, so that 2^n nodes must fail for the original node's data to be lost. The probability of single node failing by time t is $1 - (1 - c)^t$, and the probability of g nodes failing by time t is $(1 - (1 - c)^t)^g$. This leads to a completeness of $N - \sum_{i=1}^{\log_2(N)} \frac{N}{2} * (1 - (1 - c)^{i * (\frac{s}{b} + l)})^{2^{i-1}}$. As Figures 5b and 6b show, a binomial swap forest has high completeness

in the face of failures. For example, when aggregating more than 64KB of data, a binomial swap forest loses data from an order of magnitude fewer nodes than the other techniques.

Building a binomial swap forest involves each node swapping data with $\log_2(N)$ other nodes. Assuming that failures do not impact overhead, the overhead is $N * \log_2(N)$. As Figures 5c and 6c show, the overhead of a binomial swap forest is very high (Section 4 explains how San Fermín reduces this overhead by pruning trees). Using a binomial swap forest to aggregate 1MB of data requires about 20 times more overhead than balanced trees and about 5 times more than supernodes.

Intuitively, a binomial swap forest works well for two reasons. First, bandwidth dominates when aggregating large amounts of data. Other techniques build trees with higher fan-in so that nodes contend for bandwidth, while a binomial swap forest has no contention since swaps are done with only one node at a time. Second, data is replicated widely so that failures are less likely to reduce completeness. Nodes swap repeatedly, so that an exponential number of nodes need to fail for the data to be lost.

3.3 Centralized (Direct Retrieval)

In the centralized model, a central node contacts every node, retrieves their data directly, and computes the aggregated result. The central node can eliminate almost all latency costs by pipelining the retrievals, resulting in a completion time of $l + \frac{s*N}{b}$. This is much higher than the other techniques shown in Figure 5a because the time is linear in the number of nodes and the other techniques are logarithmic. As a result, to aggregate 1MB of data using the centralized technique takes 26 days as compared to about 2 minutes with a binomial swap forest.

The completeness is the number of nodes that did not fail prior to the central node retrieving their data. The probability that a node is alive after t seconds is $(1 - c)^t$, so the expected completeness is $\sum_{i=1}^N (1 - c)^{\frac{i*s}{b} + l}$. As can be seen in Figures 5b and 6b the centralized model has very poor results, despite assuming that the central node does not fail. The poor results are because many nodes fail before they are contacted by the central node.

The overhead is the number of nodes that were alive when contacted multiplied by the data size: $\sum_{i=1}^N (1 - c)^{\frac{i*s}{b} + l} * N$. A comparison is shown in Figures 5c and 6c. These results seem fantastic for large data sizes and numbers of nodes when compared to other algorithms, however what is really happening is that many nodes fail before their data is retrieved, reducing overhead but also reducing completeness.

3.4 Balanced Trees (SDIMS)

Aggregation is often performed using trees whose internal nodes have similar degree d and whose leaf nodes have similar depth. An internal node waits for data from all of its children before computing the aggregated data and sending the aggregate result to its parent. In practice, one of the child nodes is also the parent node so only $d - 1$ children send data to the parent. The model assumes that trees are balanced and complete with degree d . If the effects of failures on completion time are ignored, the completion time is $\log_d(N) * (\frac{(d-1)*s}{b} + l)$. As Figure 5a shows, this algorithm is quite fast when the data size is small and hence latency dominates. However, the performance quickly degrades when the data size increases. Aggregating 1MB of data using a balanced tree is about 4 times slower than using a binomial swap forest.

A node that fails before sending to its parent will be missing from the result. It is also possible that both the child and parent fail after the child has sent the data, also causing the child to be missing. The completeness model captures these node failures. However, the model does not consider a cascade effect. This occurs when a parent has failed and another node is recovering the data from the children when a child fails. The node that recovers and takes the role of the child would need to recover data from the child's children. This is failure handling of a child within failure handling of the parent (a cascade effect) and is not captured in the model. In the balanced tree model, there are $\frac{N}{(d-1)*d^i}$ nodes at level i . Since there is a $\sum_{j=1}^{d-1} (1 - (1 - c)^{j*\frac{s}{b} + l})$ probability of an internal node failure with $\sum_{k=1}^{i*(d-1)} (1 - (1 - c)^{i*(\frac{(d-1)*s}{b} + l) + (k+j)*\frac{s}{b} + l})$ probability of a corresponding child failure, the balanced tree's completeness is: $N - \sum_{i=0}^{\log_d(N)-1} \frac{N}{(d-1)*d^i} * \sum_{j=1}^{d-1} (1 - (1 - c)^{j*\frac{s}{b} + l}) * (1 + \sum_{k=1}^{i*(d-1)} (1 - (1 - c)^{i*(\frac{(d-1)*s}{b} + l) + (k+j)*\frac{s}{b} + l}))$. As Figure 5b shows, the completeness is high when the aggregate data size is small. However, as the aggregate data size increases the completeness quickly falls off. When the number of nodes is varied instead (as in Figure 6b), the completeness is essentially the same as having robust internal tree nodes that are provisioned against failure. For example, with 1 million nodes it is expected that only 1% of the nodes that are excluded from the result are due to internal node failures. However, the high-degree nodes take a significant amount of time to receive the initial data from each node. The time the lowest level of internal nodes take to receive the initial data from their leaf node presents a significant time window for node failures. As a result using a binomial swap forest gives an order of magnitude improvement in completeness.

In the special case $d = 2$, the balanced tree technique

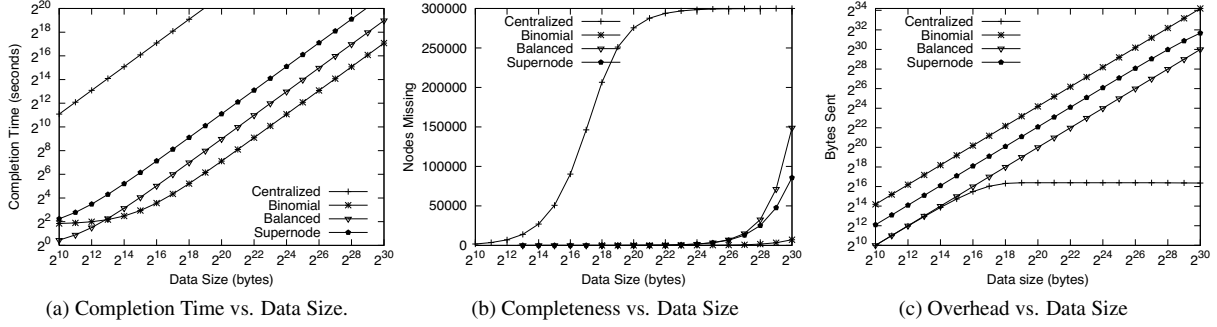


Figure 5: Scalability in the data size

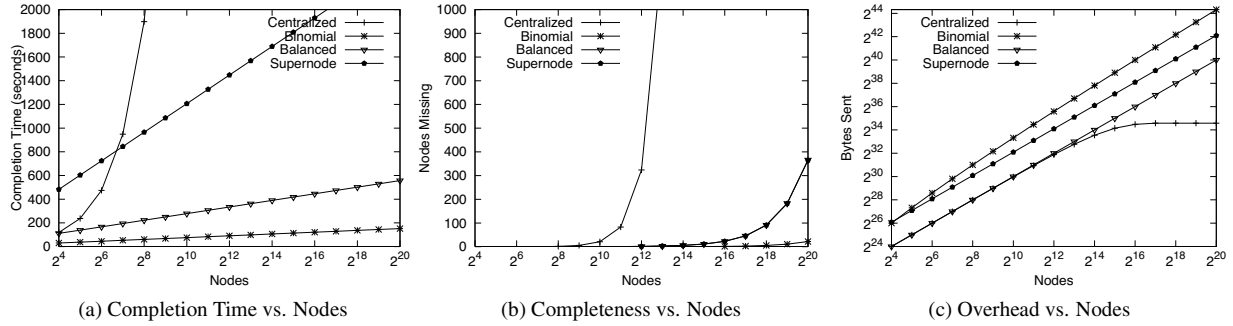


Figure 6: Scalability in the number of nodes

actually builds a binomial tree because internal nodes are counted as children at the lower levels. However, this is a single, static tree instead of a binomial swap forest. This binomial tree still has roughly four times worse completeness than using a binomial swap forest. If the degree of the balanced tree were larger (such as 16 as is used in practice), the balanced tree would have even worse completeness.

In the balanced tree model, data is only sent multiple times when failures occur. There is a base cost of N with $\sum_{i=0}^{\log_d(N)-1} \frac{N}{(d-1)*d^i}$ nodes per level and a probability of failure of $1 - (1 - c)^{\frac{(d-1)*s}{b} + l}$ with a retransmission cost of approximately $((i * (d - 1) - 1))$. The retransmission cost involves all $d - 1$ of the nodes at the prior i non-leaf levels retransmitting their aggregate data to their new parent (except the failed node). The overhead is therefore: $s * (N + \sum_{i=0}^{\log_d(N)-1} \frac{N}{(d-1)*d^i} * 1 - (1 - c)^{\frac{(d-1)*s}{b} + l} * (i * (d - 1) - 1))$ which is very respectable considering aggregate data is returned from most nodes. As Figures 5c and 6c show, the overhead is the lowest of the techniques with acceptable completeness. For example, when aggregating 1MB of data the overhead of balanced is about 4 times better than supernode and about 20 times better than using a binomial swap forest.

3.5 Supernode (Seaweed)

In this technique the nodes form a tree whose internal nodes replicate data before sending it up toward the root of the tree. Typically the tree is balanced and has uniform degree d . To prevent the loss of data when an internal node fails, there are r replicas of each internal node. When a node receives data from a child it replicates the data before replying to the child. Ideally an internal node can replicate data from a child concurrently with receiving data from another child. A node typically batches data before sending it to its parent to prevent sending small amounts of data through the tree.

The model allows internal nodes to replicate data while receiving new data, and assumes internal nodes send data to their parents as soon as they have received all data from their children. This means the model hides all but the initial delay in receiving the first bit of data ($\frac{s}{b} + l$) in the replication time ($\frac{r*d*s}{b} + 2 * l$) and leads to a completion time of $\log_d(N) * (\frac{s+r*d*s}{b} + 3 * l)$. However, the replication delay is significant as Figures 5a and 6a illustrate. Aggregating 1MB of data from 16 nodes using supernodes takes more than 8 minutes – about 16 times longer than it takes a binomial swap forest.

To simplify analysis the model assumes that there is enough replication to avoid losing all replicas of a supernode simultaneously. As a result, the only failures

that affect completeness are leaf nodes that fail before sending data to their parents. This leads to a completeness of $\sum_{i=1}^d \frac{N}{d} * (1 - c)^{i * (\frac{r}{b}) + l}$. As Figures 5b and 6b show, this delay is enough to reduce the completeness below that of the binomial swap forest (by more than an order of magnitude when aggregating 1MB). This is because in a binomial swap forest the data is replicated to exponentially many nodes, while the supernode technique has an initial significant window of vulnerability while the leaf nodes send their data to their parents.

The overhead is broken down into the cost of replicating data for internal nodes $s * \frac{(N-1)*r}{d-1}$, the cost of the leaf to internal node communication $s * (\sum_{i=1}^d \frac{r*N*(1-c)^{i*(\frac{r}{b})+l}}{d})$, and the re-replication cost $s * (\sum_{j=1}^{\lfloor \log_d(N) \rfloor - 1} \frac{N}{d^j} * (1 - (1 - c)^{j * (\frac{r*d*s+s}{b} + 3*l)})$. As Figures 5c and 6c show, the overhead of the supernode technique is better than the binomial swap forest technique by about a factor of 4 but worse than the other techniques due to the supernode replication.

4 San Fermín Details

This section describes the details of San Fermín, including an overview of the Pastry peer-to-peer (p2p) message delivery subsystem used by the San Fermín prototype, a description of how San Fermín nodes find other nodes with whom to swap, how failures are handled, how timeouts are chosen, and how trees are pruned to minimize overhead.

4.1 Pastry

Pastry [26] is a peer-to-peer system similar to Chord [28] and Tapestry [35]. Each node has a unique 160-bit nodeId that is used to identify nodes and route messages. Given a message and a destination nodeId, Pastry routes the message to the node whose nodeId is numerically closest to the destination.

Each Pastry node has two routing structures: a *routing table* and a *leaf set*. The leaf set for a node is a fixed number of nodes that have the numerically closest nodeIds to that node. This assists nodes in the last step of routing messages and in rebuilding routing tables when nodes fail.

The routing table consists of node characteristics (such as IP address, latency information, and Pastry ID) organized in rows by the length of the common prefix. When routing a message each node forwards it to the node in the routing table with the longest prefix in common with the destination nodeId.

Pastry uses nodes with nearby network proximity when constructing routing tables. As a result, the average latency of Pastry messages is less than twice the IP delay [5]. For a complete description of Pastry see the paper by Rowstron and Druschel [26].

4.2 Overview

San Fermín is part of a larger system for data aggregation. Aggregation queries are disseminated to nodes using SCRIBE [6] as the dissemination mechanism. These queries may either contain new code or references to existing code that performs two functions: extraction and aggregation. The extraction function extracts the desired data from an individual node and makes it available for aggregation. For example, if the query is over flow data, the extraction function would open the flow data logs and extract the fields of interest.

The aggregation function aggregates data from multiple nodes. This may be a simple operation like summing data items in different locations or something more complex like performing object recognition by combining data from multiple cameras.

When a node receives an aggregation request, the node disseminates the request and then runs the extraction function to obtain the data that should be aggregated. The San Fermín algorithm is used to decide how the nodes should collaborate to aggregate data. San Fermín uses the aggregation function provided in the aggregation request to aggregate data from multiple sources. Once a node has the result of the request it sends the data back to the requester. The requester then sends a *stop message* to all nodes (using SCRIBE) and they stop processing the request.

4.3 San Fermín

There are several problems that must be solved for San Fermín to work correctly and efficiently. First, a node must find other nodes with whom to swap aggregate data without complete information about the other nodes in the system. Second, a node must detect and handle the failures of other nodes. Third, a node must detect when the tree it is constructing is unlikely to be the first tree constructed and abort to reduce overhead. Each of these problems is addressed in the following subsections.

4.3.1 Finding Partners

To find nodes with whom to partner, each node first finds the longest \hat{L} its Pastry nodeId has among all nodes. This is achieved by examining the nodeIds of the nodes in its leaf set. The node first swaps with a node that has the longest \hat{L} , then the second-longest \hat{L} , and so on, until the node swaps with a node that differs in the first bit. At this point the node has built a binomial tree with aggregate data from all nodes and has computed the result.

San Fermín builds the binomial swap forest using a per-node *prefix table* that is constructed from node information in Pastry's routing table and leaf set. The i th row in the prefix table contains the nodes in \hat{L}_i from the routing table and leaf set. Each node initially swaps with a node in the highest non-empty row in its prefix table,

then swaps with nodes in successive rows until culminating with row 0. In this way San Fermín approximates binomial trees. The nodeIds are randomly distributed, so \hat{L}_p should contain about twice as many nodes as \hat{L}_{p+1} . Since nodes swap aggregate data starting at their longest \hat{L} , with each swap the number of nodes included in the aggregate data doubles. Swapping therefore doubles the number of nodes in the tree with each swap and thus approximates a binomial tree.

Swapping is a powerful mechanism for aggregating data, but there are several issues that must be addressed. Pastry only provides each node with the nodeIds for a few nodes with each \hat{L} , so how do nodes find partners with whom to swap? Also, how does a node know that another node is ready to swap with it? San Fermín solves these problems using *invitations*, which are messages delivered via Pastry that indicate that the sender is interested in swapping data with the recipient. A node only tries to swap with another node if it has previously received an invitation from that node.

In addition to sending invitations to the nodes known by Pastry, invitations are also sent to random nodeIds with the correct \hat{L} . Pastry routes these invitations to the node with the nearest nodeId. This is important because Pastry will generally only know a subset of the nodes with a given \hat{L} . To provide high completeness, a node in San Fermín must find a live node with whom to swap with each \hat{L} .

An empty row in the prefix table is handled differently depending on whether or not the associated \hat{L}_k falls within the node's leaf set. If \hat{L}_k is within the leaf set then \hat{L}_k must be empty because the Pastry leaf sets are accurate. The node skips the empty row. Otherwise, if \hat{L}_k is not within the leaf set, the node sends invitations to random nodeIds in \hat{L}_k . If no nodes exist within the \hat{L}_k the invitations will eventually time-out and the node will skip \hat{L}_k . This rarely happens, as the expected number of nodes in \hat{L}_x increases exponentially as x decreases. As an alternative to letting the invitations time-out, the nodes that receive the randomly-sent messages could respond that \hat{L}_k is empty. An empty \hat{L}_k outside of the leaf set was never observed during testing so this modification is not necessary.

4.3.2 Handling Failures

Pastry provides a failure notification mechanism that allows nodes to detect other node failures, but it has two problems that make it unsuitable for use in San Fermín. First, the polling rate for Pastry is 30 seconds, which can cause the failure of a single node to dominate the aggregation time. Second, some nodes that fail at the application level are still alive from Pastry's perspective. A node may perform Pastry functions correctly, but have some other problem that prevents it from aggregating data.

For these reasons San Fermín uses invitations to handle node failures, rather than relying exclusively on Pastry's failure notification mechanism. A node responds to an invitation to swap on a shorter \hat{L} than its current \hat{L} with a "maybe later" reply. This tells the sender that there is a live node with this \hat{L} that may later swap with it. If a "maybe later" message is not received, the node sends invitations to random nodeIds with that \hat{L} to try and locate a live node. If this fails, the node will eventually conclude the \hat{L} has no live nodes and move on to the next shorter \hat{L} .

Since timeouts are used to bypass non-responsive nodes, selecting the proper timeout period for San Fermín is important. Nodes may be overwhelmed if the timeout is too short and invitations are sent too frequently. Also short timeouts may cause nodes to be skipped during momentary network outages. If the timeout is too long then San Fermín will recover from failures slowly, increasing completion time.

Rather than having a fixed timeout length for all values of \hat{L} , San Fermín scales the timeout based on the estimated number of nodes with the value of \hat{L} . \hat{L} values with more nodes have longer timeouts because it is less likely that all the nodes will fail. Conversely, \hat{L} values with few nodes have shorter timeouts because it is more likely that all nodes will fail. In this case the node should quickly move on to the next \hat{L} if it cannot contact a live node in the current \hat{L} . A San Fermín node estimates the number of nodes in \hat{L} by estimating the density of nodes in the entire Pastry ring, which in turn is estimated from the density of nodes in its leaf set.

San Fermín sets timeouts to be a small constant t multiplied by the estimated number of nodes at \hat{L} for the given value. This means that no matter how many nodes are waiting on a group of nodes, the nodes in this group will receive fewer than $2 * t$ invitations per second, on average. This timeout rate also keeps the overhead from invitations low.

4.3.3 Pruning Trees

Each San Fermín node builds its own tree to improve performance and tolerate failures, but only one tree will win the race to compute the final result. If San Fermín knew the winner in advance it could build only the winning tree and avoid the overhead of building the losing trees. Instead, San Fermín builds all trees and prunes those unlikely to win. San Fermín prunes a tree whenever its root node cannot find another node with whom to swap but there exists a live node with that \hat{L} value. This is accomplished by the use of "no" responses to invitations.

A node sends a "no" response to an invitation when its current \hat{L} is shorter than the \hat{L} contained in the invitation. This means the node receiving the invitation has already aggregated the data in the \hat{L} and has no need to swap

with the node that sent the invitation. Whenever a node receives a “no” response it does not send future invitations to the node that sent the response. Unlike a “maybe later” response, “no” responses do not reset the timeout. If a node that has received a “no” response and it cannot find a partner for this value of \hat{L} before the timeout expires, the node simply aborts its aggregation.

Note that a node will only receive a “no” response when two other nodes have its data in their aggregate data. This is because the node that sends a “no” response must have already aggregated data for that \hat{L} (and therefore must already have the inviting node’s data). Since the node that sent the “no” response has aggregated data for the \hat{L} via a swap then another node must also have the inviting node’s data.

4.3.4 San Fermín Pseudocode

This section presents pseudocode for the San Fermín algorithm, omitting details of error and timeout handling.

```
When a node receives a message:
  If message is an invitation:
    If current  $\hat{L}$  shorter than  $\hat{L}$  in invitation
      reply with no
    else reply with maybe_later and
      remember node that sent invitation
  If message is a no, remember that one was received
  If message is a maybe_later then reset time-out
  If message is a stop then stop aggregation

# Called to begin aggregation
Function aggregate_data(data, requester):
  Initialize the prefix_table from Pastry tables
  for  $\hat{L}$  in prefix_table from long to short:
    Call aggregate_ $\hat{L}$  to swap data with a node
    If swap successful
      compute aggregation of existing and received data
  Send aggregate data (the result) to the requester

# A helper function to do aggregation for a value of  $\hat{L}$ 
Function aggregate_ $\hat{L}$ (data, known_nodes):
  Try to swap data with nodes with this  $\hat{L}$  from whom
    an invitation was received
  If successful then return the aggregate data
  Send invitations to nodes in prefix table with this  $\hat{L}$ 
  While waiting for a time-out:
    If a node connects, swap with it and return the data
    Try to swap with nodes from whom we got invitations
    If success then return the aggregate data

# Time-out
if we got a no message, then stop (do not return)
otherwise return no aggregate data
```

5 Evaluation

This section answers several questions about San Fermín:

- How does San Fermín compare to other existing solutions?
- How well does San Fermín scale with the number of nodes and the data size?
- How well does San Fermín tolerate failures?
- What is the overhead of San Fermín?

- How effective is San Fermín at utilizing high-capacity nodes?

5.1 Comparison

We developed a Java-based San Fermín prototype that runs on the Java FreePastry implementation on Planet-Lab [23]. The SDIMS prototype (which also runs on FreePastry) was compared against San Fermín in several experiments using randomly-selected live nodes with transitive connectivity and clock skew of less than 1 second. All experiments for a particular number of nodes used the same set of nodes.

The comparison with SDIMS demonstrates that existing techniques are inadequate for aggregating large amounts of data. SDIMS was designed for streaming small amounts of data whereas San Fermín is designed for one-shot queries of large amounts of data. Ideally, large SDIMS data would be treated as separate attributes and aggregated up separate trees. However, since this may include only part of a node’s data, this may skew the distribution of results returned. Therefore all data is aggregated as a single attribute.

One complication with comparing the two is zombie nodes in Pastry. San Fermín uses timeouts to identify quickly nodes that are unresponsive. SDIMS however, relies on the underlying p2p network to identify unresponsive nodes, leaving it vulnerable to zombie nodes. After consulting with the SDIMS authors, we learned that they avoid this issue on PlanetLab by building more than one tree (typically four) and using the aggregate data from the first tree to respond. In the experiments we measured SDIMS using both one tree (SDIMS-1) and four trees (SDIMS-4).

The experiments compare the time, overhead and completeness of SDIMS and San Fermín. A small amount of accounting information was included in the aggregate data for determining which nodes’ data were included in the result. Unless specified otherwise, each experiment used 100 nodes and aggregated 1MB from each node, each data point is the average of 10 runs, and the error bars represent 1 standard deviation. All tests were limited to 5 minutes. In SDIMS the aggregate data trickles up to the root over time, so the SDIMS result was considered complete when either the aggregate data from all nodes reached the root or the aggregate data from at least half the nodes reached the root and no new data were received in 20 seconds.

Different aggregation functions such as summing counters, comparison for equals, maximum, and string parsing were experimented with. The choice of aggregation function did not have any noticeable effect on the experiments.

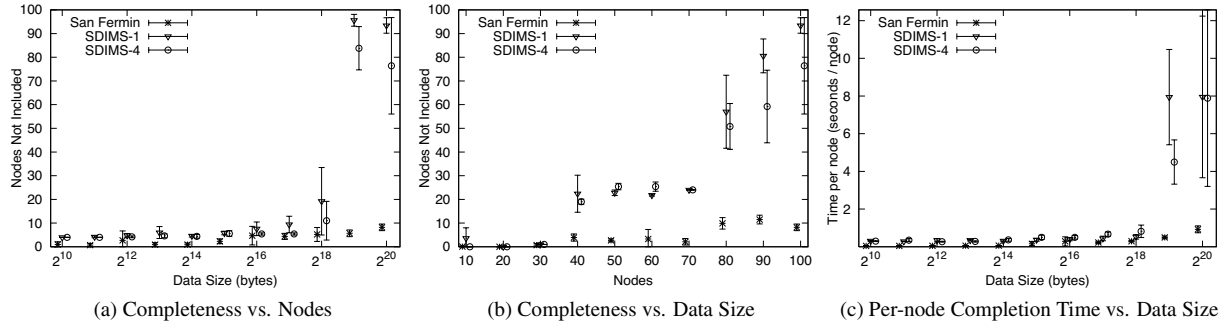


Figure 7: Comparison of San Fermín and SDIMS on PlanetLab. SDIMS-1 is SDIMS using a single tree; SDIMS-4 is SDIMS using four trees.

5.1.1 Completeness

The first set of PlanetLab experiments measures completeness as the aggregated data size increases (Figure 7a). The number of nodes not included in the aggregate data is small for each algorithm until the data size exceeds 256KB. At that point SDIMS performs poorly because high-degree internal nodes are overwhelmed (shown in more detail in Section 5.4). San Fermín continues to include the aggregate data from most nodes.

The next set of experiments measures how the number of nodes affects completeness (Figure 7b). When there are few nodes SDIMS-4 and San Fermín algorithms do quite well. Once there are more than 30 nodes the SDIMS trees perform poorly due to high-degree internal nodes being overwhelmed with traffic.

5.1.2 Completion Time

Figure 7c shows per-node completion time, which is the completion time of the entire aggregation divided by the number of nodes whose data is included in the result. This metric allows for meaningful comparisons between San Fermín and SDIMS because they may produce results with different completeness. Data sizes larger than 256KB significantly increases the per-node completion time of SDIMS, while San Fermín increases only slightly. Although not shown, for a given data size the number of nodes has little effect on the per-node completion time.

Figure 8 illustrates the performance of individual aggregations in terms of both completion time and completeness. Points near the origin have low completion time and high completeness, and are thus better than points farther away. San Fermín’s points are clustered near the origin, indicating that it consistently provides high completeness and low completion time even in a dynamic environment like PlanetLab. SDIMS’s performance is highly variable — SDIMS-1 occasionally has very high completeness and low completion time, but more often performs poorly with more than half the ag-

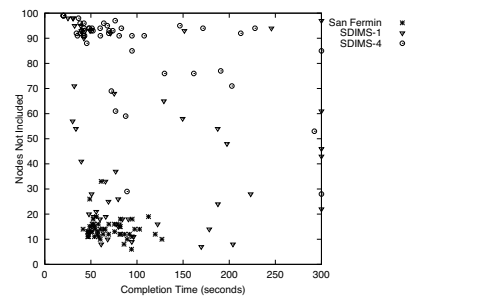


Figure 8: Completeness and Completion time of San Fermín and SDIMS on PlanetLab. Each point represents a single run. Points near the origin are better because they have lower completion time and higher completeness.

gregations missing at least 35 nodes from the result. SDIMS-4 performs even worse with all but 10 aggregations missing at least 80 nodes.

5.2 Scalability

We used a simulator to measure the scalability of San Fermín beyond that possible on PlanetLab. The simulator is event-driven and based on measurements of real network topologies. Several simplifications were made to improve scalability and reduce the running time: global knowledge is used to construct the Pastry routing tables; the connection teardown states of TCP are not modeled (as San Fermín does not wait for TCP to complete the connection closure); and lossy network links are not modeled.

The simulations used network topologies from the University of Arizona’s Department of Computer Science (CS) and PlanetLab. The CS topology consists of a central switch connected to 142 systems with 1 Gbps links, 205 systems with 100 Mbps links, and 6 legacy systems with 10 Mbps links. Simulations using fewer nodes were constructed by randomly choosing nodes from the entire set.

The PlanetLab topology was derived from data provided by the S^3 project [32]. The data provides pairwise

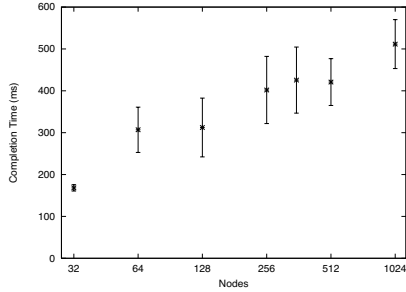


Figure 9: Completion Time vs. Nodes, CS Topology.

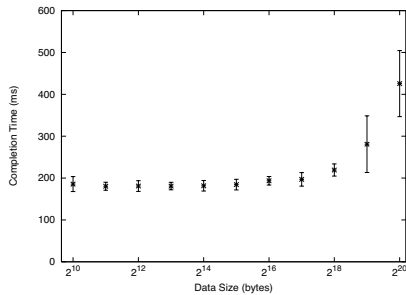


Figure 10: Completion Time vs. Data Size, CS Topology. Each experiment used all 353 nodes.

latency and bandwidth measurements for all nodes on PlanetLab. Intra-site topologies were assumed to consist of a single switch connected to all nodes. The latency of an intra-site link was set to 1/2 of the minimum latency seen by the node on that link, and the bandwidth to the maximum bandwidth seen by the node. Inter-site latencies were set to the minimum latency between the two sites as reported by S^3 minus the intra-site latencies of the nodes. The inter-site bandwidths were set to the maximum bandwidths between the two sites.

In both topologies the Pastry nodeIds were randomly assigned, and a different random seed was used for each simulation. As in the PlanetLab experiments, unless specified otherwise, each experiment used 100 nodes and aggregated 1MB of data from each node, each data point is the average of 10 runs, and the error bars represent 1 standard deviation.

The first experiment varied the number of nodes in the system to demonstrate the scalability of San Fermín; the results of the CS topology are shown in Figure 9. The completion time increases slightly as the number of nodes increases; when the number of nodes increases from 32 nodes to 1024 nodes the completion time only increases by about a factor of four. A 1024 node aggregation of 1MB completed in under 500ms. The PlanetLab topology (not shown) has similar behavior — the completion time also increases by approximately a factor of four as the number of nodes increases from 32 to 1024.

Figure 10 shows the result of varying the data size

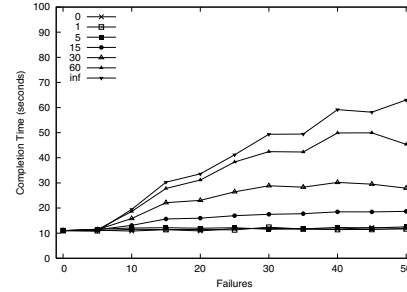


Figure 11: Completion Time vs. Failures, PlanetLab Topology. Each curve represents a different Pastry convergence time, from 0 seconds to infinity.

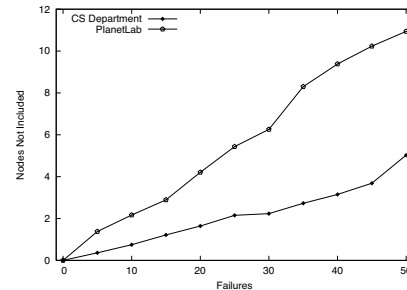


Figure 12: Completeness vs. Failures.

while using all 353 nodes in the CS topology. The completion time is dominated by the p2p and message header overheads for data sizes under 128KB. When aggregating more than 128KB the completion time increases significantly. The PlanetLab topology (not shown) has a similar pattern in which all of the data sizes under 128KB take about 4 seconds and thereafter the mean time increases linearly with the data size.

In all experiments the result included data from all nodes, therefore completeness results are not presented.

5.3 Failure Handling

The next set of simulations measured the effectiveness of San Fermín at tolerating node failures. Failure traces were synthetically generated by randomly selecting nodes to fail during the aggregation. The times of the failures were chosen randomly from the start time of the aggregation to the original completion time. The p2p time to notice failures is varied to demonstrate the effect on San Fermín.

The timeout mechanism in San Fermín allows it to detect failures before the underlying p2p does. As a result, the average completion time is less than the Pastry recovery time (Figure 11). On the PlanetLab topology, when the Pastry recovery time is less than 5 seconds, the cost of failures is negligible because other nodes use the time to aggregate the remaining data (leaving only failed subtrees to complete). When the recovery time is more than 5 seconds then some nodes end up timing-out

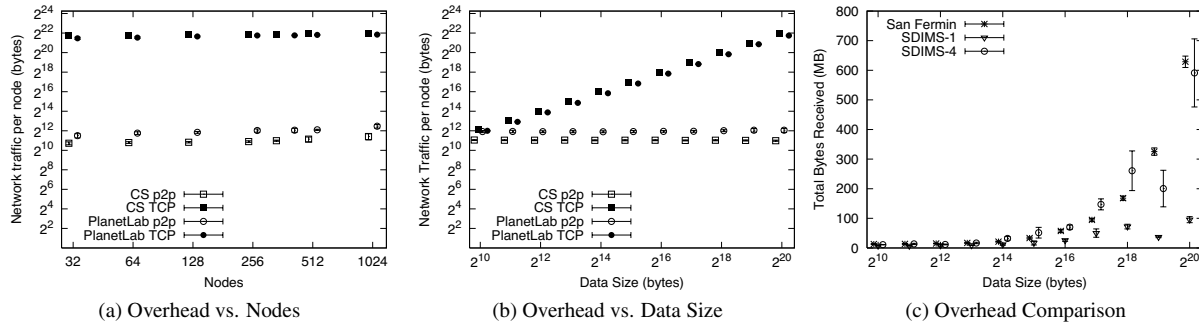


Figure 13: San Fermín Overhead. Overhead is segregated into p2p and TCP traffic for (a) and (b).

a failed subtree before continuing. The CS department topology (not depicted) typically completes in less than 500ms so all non-zero Pastry recovery times increase the completion time. However, the average completion time is less than the Pastry recovery time for all recovery times greater than 1 second.

Figure 12 shows how failures affect completeness. Since failures occurred over the original aggregation time, altering the Pastry convergence time has little effect on the completeness (and so the average of all runs is shown). The number of failures has different effects on the PlanetLab and CS topologies. There is greater variability of link bandwidths in the PlanetLab topology, which causes swaps to happen more slowly in some subtrees. Failures in those trees are more likely to decrease completeness than in the CS topology, which has more uniform link bandwidths and the data swaps happen more quickly. In both topologies the completeness is better than the number of nodes that failed — in most cases a node fails after enough swaps have occurred to ensure its data is included in the result.

5.4 Overhead

In this section two aspects of overhead are examined: the cost of invitations and the overhead characteristics as measured on PlanetLab. The two characteristics of interest are the total traffic during aggregation and the peak traffic observed by a node.

5.4.1 Overhead Composition

We ran simulations with varying numbers of nodes on the CS and PlanetLab network topologies to evaluate the composition of network traffic from San Fermín (Figure 13a). The traffic is segregated by type (p2p or TCP). The p2p traffic is essentially the traffic from invitations and responses while the TCP traffic is from nodes swapping aggregate data. The traffic per node does not substantially increase as the number of nodes increases, meaning that the total traffic is roughly linear in the number of nodes.

San Fermín on the PlanetLab topology has higher p2p

and lower TCP traffic than on the CS topology. This is because PlanetLab’s latency is higher and more variable, causing the overall aggregation process to take much longer (which naturally increases the number of p2p messages sent). The PlanetLab bandwidth is also highly variable (especially intra-site links versus inter-site links). This causes high variability in partnering time, so that slow partnerings that might otherwise occur do not because faster nodes have already computed the result.

As Figure 13a demonstrates, the p2p traffic is insignificant when 1MB of data is aggregated. Figure 13b shows how the composition of p2p and TCP traffic varies as the data size is varied. This is important for two reasons. First, it shows that the p2p traffic does not contribute significantly to the total overhead. Second, it shows how the total overhead varies with the data size. Doubling the data size caused the total overhead to roughly double.

Another notable result is that the standard deviations were quite small, less than 4% in all cases. This makes it difficult to discern the error bars in the figures.

5.4.2 Total Traffic

The total network traffic of San Fermín was also measured experimentally on PlanetLab (Figure 13c). The results from SDIMS are presented for comparison. For less than 256KB, SDIMS-1 incurs the least overhead, followed by San Fermín and then SDIMS-4. After 256KB the overhead for SDIMS actually decreases because the completeness decreases. Nodes are overwhelmed by traffic and fail. A single internal node failure causes the loss of all data for it and its children until either the internal node recovers or the underlying p2p network converges.

5.4.3 Peak Node Traffic

The peak traffic experienced by a node is important because it can overload a node (Figure 14). To evaluate peak node traffic, an experiment was run on PlanetLab with 30 nodes aggregating 1 MB of data (30 nodes being

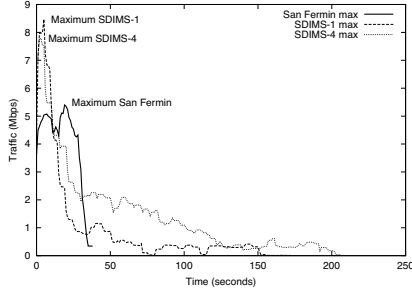


Figure 14: Peak Node Traffic. Each data point represents the peak traffic experienced by a node during that second of the aggregation.

the most nodes for which SDIMS had high completeness).

SDIMS internal nodes may receive data from many of their children simultaneously; the large initial peak of SDIMS traffic causes internal nodes that are not well-provisioned to either become zombies or fail. On the other hand, San Fermín nodes only receive data from one partner at a time, reducing the maximum peak traffic. As a result, San Fermín has a maximum peak node traffic that is less than 2/3 that of SDIMS.

5.5 Capacity

An important aspect of San Fermín is that each node creates its own binomial aggregation tree. By racing to compute the aggregate data, high-capacity nodes naturally fill the internal nodes of the binomial trees, while low-capacity nodes fill the leaves and ultimately prune their own aggregation trees.

The final experiment measures how effective San Fermín is at pruning low-capacity nodes. 1MB of data was aggregated from 100 PlanetLab nodes 10 times. The state of each node was recorded when the aggregation completed. Table 2 shows the results, including the number of swaps remaining for each node to complete its aggregation and the average peak bandwidth of nodes with the same number of swaps remaining. Nodes with the higher capacity had fewer swaps remaining, whereas the nodes with lower capacity pruned their trees. The nodes in the middle tended to prune their trees but some were still working; the average peak bandwidth of these nodes was 2.1Mbps, whereas the average peak bandwidth of the nodes still working was 3.2Mbps. This means that nodes that are pruned have about 1/3 less observed capacity than those nodes that are still aggregating data. This illustrates that San Fermín is effective at having high-capacity nodes perform the aggregation and having low-capacity nodes prune their trees.

6 Related Work

Using trees to aggregate data from distributed nodes is not a new idea. The seminal work of Chang on

Remaining Swaps	Pruned Nodes		Working Nodes	
	Number	Mbps	Number	Mbps
0	0	0.0	38	4.3
1	0	0.0	105	3.9
2	0	0.0	116	3.6
3	9	2.5	56	2.3
4	82	2.0	32	2.2
5	143	2.0	19	1.2
6	107	2.4	9	1.1
7	62	2.0	1	0.8
8	14	1.7	0	0.0
9	16	2.4	0	0.0
10	3	1.6	0	0.0
11	0	0	0	0.0
12	2	1.9	0	0.0

Table 2: Effectiveness of San Fermín at using high-capacity nodes. The *number* column is the number of nodes with the given number of swap remaining when the aggregation completed; the *Mbps* column is the average peak bandwidth of those nodes.

Echo-Probe [7] formulated polling distant nodes and collecting data as a graph theory problem. More recently, Willow [30], SOMO [34], DASIS [1], Cone [3], SDIMS [31], Ganglia [21], and PRISM [15], have used trees to aggregate attributes Willow, SOMO, and Ganglia use one tree for all attributes, whereas SDIMS, Cone, and PRISM use one tree per attribute.

Seaweed [22] performs one-shot queries of small amounts of data and like San Fermín is focused on completeness. However, Seaweed trades completion time for completeness in that queries are expected to live for many hours or even days as nodes come online and return aggregate data. Seaweed uses a supernode-based solution that further delays the timeliness of the initial aggregate data. Instead San Fermín focuses on a different part of the design space, robustly returning aggregate data from existing nodes in a timely manner.

CONCAST [4] implements many-to-one channels as a network service. It uses routers to aggregate data over a single tree. As the size of the aggregate data grows the memory and processing requirements on routers becomes prohibitive.

Gossip and epidemic protocols have also been used for aggregation [18, 13, 17, 16], including Astrolabe [29]. Unstructured protocols that rely on random exchanges face a trade-off between precision and scalability. Structured protocols, such as Astrolabe, impose a structure on the data exchanges that prevents duplication. This is at the cost of creating and maintaining a structure, and confining the data exchanges to adhere to the structure.

Data aggregation is also an issue in sensor networks. Unlike San Fermín, the major concerns in sensor networks are power consumption and network traffic. Examples of data aggregation in sensor networks are

TAG [20], Hourglass [27], and Cougar [33].

Distributed query processing involves answering queries across a set of distributed nodes. The most relevant to our work are systems such as PIER [14], which stores tuples in a DHT as part of processing a query. Distributed query processing also encompasses performing queries on continuous streams of data, as is done in Aurora [8], Medusa [8], and HiFi [11].

There are several systems that have focused on aggregating data from large data sets from a programming language perspective [10, 24]. However neither system focuses on sending large amounts data over the network.

7 Conclusions

This paper presents San Fermín, a technique for aggregating large amounts of data that when aggregating 1MB of data provides 2-6 times better completeness than SDIMS, at 61-76% of the completion time, and with better scalability characteristics. San Fermín has a peak node traffic more than 1/3 lower than that of SDIMS, which accounts for much of the higher completeness. Our analysis shows that when 10% of the nodes fail during aggregation San Fermín still computes the aggregated result from 97% of the nodes. San Fermín also scales well with the number of nodes or the data size – completion time increases by less than a factor of 4 if the number of nodes increases from 32 to 1024, and by about a factor of 2 as the data size increases from 256KB to 1MB.

Acknowledgments

We would like to thank the SDIMS group (especially Navendu Jain) for helping us use and evaluate SDIMS. We would especially like to thank our shepherd Arun Venkataramani and the anonymous reviewers for their helpful feedback.

References

- [1] K. Albrecht, R. Arnold, M. Gähwiler, and R. Wattenhofer. Aggregating information in peer-to-peer systems for improved join and leave. In *Peer-to-Peer Computing*, 2004.
- [2] PlanetLab - All Sites Ping. <http://ping.ececs.uc.edu/ping/>.
- [3] R. Bhagwan, G. Varghese, and G. Voelker. Cone: Augmenting DHTs to support distributed resource discovery. Technical Report CS2003-0755, UCSD, 2003.
- [4] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: design and implementation of an active network service. *IEEE JSAC*, 19(3), 2001.
- [5] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *FuDiCo*, 2002.
- [6] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), 2002.
- [7] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE TSE*, 1982.
- [8] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.
- [9] M. Collins. Personal correspondence, Sept. 2006.
- [10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [11] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, pages 290–304, 2005.
- [12] J. Guicahrd, F. le Faucheur, and J. P. Vasseur. *Definitive MPLS Network Designs*. Cisco Press, 2005.
- [13] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *IEEE DSN*, 2001.
- [14] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.
- [15] N. Jain, D. Kit, D. Mahajan, M. Dahlin, and Y. Zhang. PRISM: Precision integrated scalable monitoring. Technical Report TR-06-22, University of Texas, Feb. 2006.
- [16] M. Jelasity, W. Kowalczyk, and M. van Steen. An approach to massively distributed aggregate computing on peer-to-peer networks. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2004.
- [17] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *ICDCS*, 2004.
- [18] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM TOCS*, 23(3):219–252, 2005.
- [19] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.
- [20] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *OSDI*, 2002.
- [21] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganga distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.
- [22] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with Seaweed. In *VLDB*, 2006.
- [23] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *HotNets*, 2002.
- [24] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
- [25] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
- [26] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *ICDCS*, 2001.
- [27] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical Report TR-21-04, Harvard University, 2004.
- [28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for Internet applications. In *SIGCOMM*, 2001.
- [29] R. van Renesse and K. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, May 2003.
- [30] R. van Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.
- [31] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.
- [32] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. S3: a scalable sensing service for monitoring large networked systems. In *SIGCOMM workshop on Internet network management*, 2006.
- [33] Y. Yao and J. Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD*, 31(3):9–18, Sept. 2002.
- [34] Z. Zhang, S.-M. Shi, and J. Zhu. SOMO: Self-organized metadata overlay for resource management in P2P DHT. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.
- [35] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE JSAC*, 2003.

Remus: High Availability via Asynchronous Virtual Machine Replication

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer,
Mike Feeley, Norm Hutchinson, and Andrew Warfield*

*Department of Computer Science
The University of British Columbia
{brendan, geoffrey, dmeyer, feeley, norm, andy}@cs.ubc.ca*

Abstract

Allowing applications to survive hardware failure is an expensive undertaking, which generally involves re-engineering software to include complicated recovery logic as well as deploying special-purpose hardware; this represents a severe barrier to improving the dependability of large or legacy applications. We describe the construction of a general and transparent high availability service that allows existing, *unmodified* software to be protected from the failure of the physical machine on which it runs. *Remus* provides an extremely high degree of fault tolerance, to the point that a running system can transparently continue execution on an alternate physical host in the face of failure with only seconds of downtime, while completely preserving host state such as active network connections. Our approach encapsulates protected software in a virtual machine, asynchronously propagates changed state to a backup host at frequencies as high as forty times a second, and uses speculative execution to concurrently run the active VM slightly ahead of the replicated system state.

1 Introduction

Highly available systems are the purview of the very rich and the very scared. However, the desire for reliability is pervasive, even among system designers with modest resources.

Unfortunately, high availability is hard — it requires that systems be constructed with redundant components that are capable of maintaining and switching to backups in the face of failure. Commercial high availability systems that aim to protect modern servers generally use specialized hardware, customized software, or both (e.g [12]). In each case, the ability to transparently survive failure is complex and expensive enough to prohibit deployment on common servers.

This paper describes *Remus*, a software system that provides OS- and application-agnostic high availability on commodity hardware. Our approach capitalizes on the ability of virtualization to migrate running VMs between physical hosts [6], and extends the technique to replicate snapshots of an entire running OS instance at very high frequencies — as often as every 25ms — between a pair of physical machines. Using this technique, our system discretizes the execution of a VM into a series of replicated snapshots. External output, specifically transmitted network packets, is not released until the system state that produced it has been replicated.

Virtualization makes it possible to create a copy of a running machine, but it does not guarantee that the process will be efficient. Propagating state synchronously at every change is impractical: it effectively reduces the throughput of memory to that of the network device performing replication. Rather than running two hosts in lock-step [4] we allow a single host to execute *speculatively* and then checkpoint and replicate its state *asynchronously*. System state is not made externally visible until the checkpoint is committed — we achieve high-speed replicated performance by effectively running the system tens of milliseconds in the past.

The contribution of this paper is a practical one. Whole-system replication is a well-known approach to providing high availability. However, it usually has been considered to be significantly more expensive than application-specific checkpointing techniques that only replicate relevant data [15]. Our approach may be used to bring HA “to the masses” as a platform service for virtual machines. In spite of the hardware and software constraints under which it operates, this system provides protection equal to or better than expensive commercial offerings. Many existing systems only actively mirror persistent storage, requiring applications to perform recovery from crash-consistent persistent state. In contrast, *Remus* ensures that regardless of the moment at which the primary fails, no externally visible state is ever lost.

*also of Citrix Systems, Inc.

1.1 Goals

Remus aims to make mission-critical availability accessible to mid- and low-end systems. By simplifying provisioning and allowing multiple servers to be consolidated on a smaller number of physical hosts, virtualization has made these systems more popular than ever. However, the benefits of consolidation come with a hidden cost in the form of increased exposure to hardware failure. Remus addresses this by commodifying high availability as a service offered by the virtualization platform itself, providing administrators of individual VMs with a tool to mitigate the risks associated with virtualization.

Remus's design is based on the following high-level goals:

Generality. It can be prohibitively expensive to customize a single application to support high availability, let alone the diverse range of software upon which an organization may rely. To address this issue, high availability should be provided as a low-level service, with common mechanisms that apply regardless of the application being protected or the hardware on which it runs.

Transparency. The reality in many environments is that OS and application source may not even be available to modify. To support the broadest possible range of applications with the smallest possible barrier to entry, high availability should not require that OS or application code be modified to support facilities such as failure detection or state recovery.

Seamless failure recovery. No externally visible state should ever be lost in the case of single-host failure. Furthermore, failure recovery should proceed rapidly enough that it appears as nothing more than temporary packet loss from the perspective of external users. Established TCP connections should not be lost or reset.

These are lofty goals, entailing a degree of protection well beyond that provided by common HA systems, which are based on asynchronous storage mirroring followed by application-specific recovery code. Moreover, the desire to implement this level of availability *without* modifying the code within a VM necessitates a very coarse-grained approach to the problem. A final and pervasive goal of the system is that it realize these goals while providing deployable levels of performance even in the face of SMP hardware that is common on today's server hardware.

1.2 Approach

Remus runs paired servers in an active-passive configuration. We employ three major techniques in order to overcome the difficulties traditionally associated with this approach. First, we base our system on a virtualized infrastructure to facilitate whole-system replication. Second,

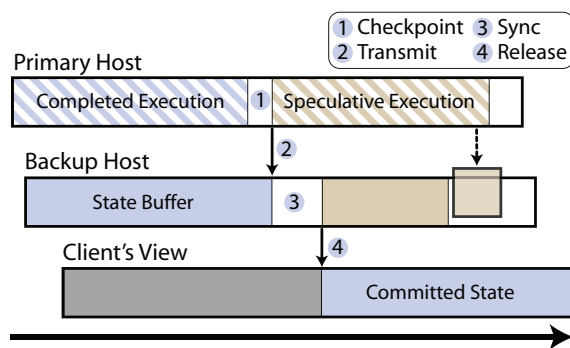


Figure 1: Speculative execution and asynchronous replication in Remus.

we increase system performance through speculative execution, which decouples external output from synchronization points. This allows the primary server to remain productive, while synchronization with the replicated server is performed asynchronously. The basic stages of operation in Remus are given in Figure 1.

VM-based whole-system replication. Hypervisors have been used to build HA systems before [4]. In that work, virtualization is used to run a pair of systems in lock-step, and additional support has been added to ensure that VMs on a pair of physical hosts follow a deterministic path of execution: external events are carefully injected into both the primary and fallback VMs so that they result in identical states. Enforcing such deterministic execution suffers from two fundamental problems. First, it is highly architecture-specific, requiring that the system have a comprehensive understanding of the instruction set being executed and the sources of external events. Second, it results in an unacceptable overhead when applied in multi-processor systems, where shared-memory communication between processors must be accurately tracked and propagated [8].

Speculative execution. Replication may be achieved either by copying the state of a system or by replaying input deterministically. We believe the latter to be impractical for real-time operation, especially in a multi-processor environment. Therefore, Remus does not attempt to make computation deterministic — there is a very real possibility that the output produced by a system after a given checkpoint will be different if the system is rolled back to that checkpoint and its input is replayed. However, the state of the replica needs to be synchronized with the primary only when the output of the primary has become externally visible. Instead of letting the normal output stream dictate when synchronization must occur, we can buffer output¹ until a more convenient time, performing computation *speculatively* ahead of synchronization points. This allows a favorable trade-off to be made between output latency and runtime over-

head, the degree of which may be controlled by the administrator.

Asynchronous replication. Buffering output at the primary server allows replication to be performed *asynchronously*. The primary host can resume execution at the moment its machine state has been captured, without waiting for acknowledgment from the remote end. Overlapping normal execution with the replication process yields substantial performance benefits. This enables efficient operation even when checkpointing at intervals on the order of tens of milliseconds.

2 Design and Implementation

Figure 2 shows a high-level view of our system. We begin by encapsulating the machine to be protected within a VM. Our implementation is based on the Xen virtual machine monitor [2], and extends Xen’s support for live migration to provide fine-grained checkpoints. An initial subset of our checkpointing support has been accepted into the upstream Xen source.

Remus achieves high availability by propagating frequent checkpoints of an *active* VM to a *backup* physical host. On the backup, the VM image is resident in memory and may begin execution immediately if failure of the active system is detected. Because the backup is only periodically consistent with the primary, all network output must be buffered until state is synchronized on the backup. When a complete, consistent image of the host has been received, this buffer is released to external clients. The checkpoint, buffer, and release cycle happens very frequently – we include benchmark results at frequencies up to forty times per second, representing a whole-machine checkpoint including network and on-disk state every 25 milliseconds.

Unlike transmitted network traffic, disk state is not externally visible. It must, however, be propagated to the remote host as part of a complete and consistent snapshot. To maintain disk replication, all writes to the primary disk are transmitted asynchronously to the backup, where they are buffered in RAM until the corresponding memory checkpoint has arrived. At that point, the complete checkpoint is acknowledged to the primary, which then releases outbound network traffic, and the buffered disk writes are applied to the backup disk.

It is worth emphasizing that the virtual machine does not actually execute on the backup host until a failure occurs. It simply acts as a receptacle for checkpoints of the active VM. This consumes a relatively small amount of the backup host’s resources, allowing it to concurrently protect VMs running on multiple physical hosts in an N-to-1-style configuration. Such a configuration gives administrators a high degree of freedom to balance the degree of redundancy against resource costs.

2.1 Failure model

Remus provides the following properties:

1. The fail-stop failure of any single host is tolerable.
2. Should both the primary and backup hosts fail concurrently, the protected system’s data will be left in a crash-consistent state.
3. No output will be made externally visible until the associated system state has been committed to the replica.

Our goal is to provide completely transparent recovery from fail-stop failures of a single physical host. The compelling aspect of this system is that high availability may be easily retrofitted onto existing software running on commodity hardware. It uses a pair of commodity host machines, connected over redundant gigabit Ethernet connections, and survives the failure of any one of these components. By incorporating block devices into its state replication protocol, it avoids requiring expensive, shared network-attached storage for disk images.

We do not aim to recover from software errors or non-fail-stop conditions. As observed in [5], this class of approach provides complete system state capture and replication, and as such will propagate application errors to the backup. This is a necessary consequence of providing both transparency and generality.

Our failure model is identical to that of commercial HA products, which provide protection for virtual machines today [31, 30]. However, the degree of protection offered by these products is substantially less than that provided by Remus: existing commercial products respond to the failure of a physical host by simply rebooting the VM on another host from its crash-consistent disk state. Our approach survives failure on time frames similar to those of live migration, and leaves the VM running and network connections intact. Exposed state is not lost and disks are not corrupted.

2.2 Pipelined Checkpoints

Checkpointing a running virtual machine many times per second places extreme demands on the host system. Remus addresses this by aggressively pipelining the checkpoint operation. We use an epoch-based system in which execution of the active VM is bounded by brief pauses in execution in which changed state is atomically captured, and external output is released when that state has been propagated to the backup. Referring back to Figure 1, this procedure can be divided into four stages: (1) Once per epoch, pause the running VM and copy any changed state into a buffer. This process is effectively the stop-and-copy stage of live migration [6], but as described

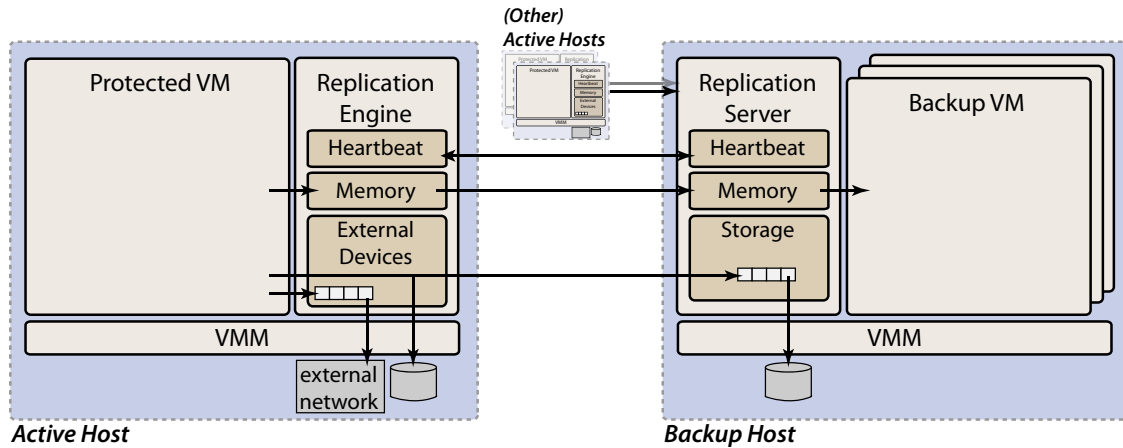


Figure 2: Remus: High-Level Architecture

later in this section it has been dramatically optimized for high-frequency checkpoints. With state changes preserved in a buffer, the VM is unpaused and speculative execution resumes. (2) Buffered state is transmitted and stored in memory on the backup host. (3) Once the complete set of state has been received, the checkpoint is acknowledged to the primary. Finally, (4) buffered network output is released.

The result of this approach is that execution is effectively discretized at checkpoint boundaries; the acknowledgment of a completed checkpoint by the backup triggers the release of network traffic that has been buffered and represents an atomic transition into the new epoch.

2.3 Memory and CPU

Checkpointing is implemented above Xen's existing machinery for performing live migration [6]. Live migration is a technique by which a virtual machine is relocated to another physical host with only slight interruption in service. To accomplish this, memory is copied to the new location while the VM continues to run at the old location. During migration, writes to memory are intercepted, and dirtied pages are copied to the new location in rounds. After a specified number of intervals, or when no forward progress is being made because the virtual machine is writing to memory at least as fast as the migration process can copy it out, the guest is suspended and the remaining dirty memory is copied out along with the current CPU state. At this point the image on the new location is activated. Total downtime depends on the amount of memory remaining to be copied when the guest is suspended, but is typically under 100ms. Total migration time is a function of the amount of memory in use by the guest, and its *writable working set* [6], which

is the set of pages changed repeatedly during guest execution.

Xen provides the ability to track guest writes to memory using a mechanism called *shadow page tables*. When this mode of operation is enabled, the VMM maintains a private ("shadow") version of the guest's page tables and exposes these to the hardware MMU. Page protection is used to trap guest access to its internal version of page tables, allowing the hypervisor to track updates, which are propagated to the shadow versions as appropriate.

For live migration, this technique is extended to transparently (to the guest) mark all VM memory as read only. The hypervisor is then able to trap all writes that a VM makes to memory and maintain a map of pages that have been dirtied since the previous round. Each round, the migration process atomically reads and resets this map, and the iterative migration process involves chasing dirty pages until progress can no longer be made. As mentioned above, the live migration process eventually suspends execution of the VM and enters a final "stop-and-copy" round, where any remaining pages are transmitted and execution resumes on the destination host.

Remus implements checkpointing as repeated executions of the final stage of live migration: each epoch, the guest is paused while changed memory and CPU state is copied to a buffer. The guest then resumes execution on the current host, rather than on the destination. Several modifications to the migration process are required in order to provide sufficient performance and to ensure that a consistent image is always available at the remote location. These are described below.

Migration enhancements. In live migration, guest memory is iteratively copied over a number of rounds and may consume minutes of execution time; the brief service interruption caused by the singular stop-and-copy

phase is not a significant overhead. This is not the case when capturing frequent VM checkpoints: *every* checkpoint is just the final stop-and-copy phase of migration, and so this represents a critical point of optimization in reducing checkpoint overheads. An examination of Xen's checkpoint code revealed that the majority of the time spent while the guest is in the suspended state is lost to scheduling, largely due to inefficiencies in the implementation of the xenstore daemon that provides administrative communication between guest virtual machines and domain 0.

Remus optimizes checkpoint signaling in two ways: First, it reduces the number of inter-process requests required to suspend and resume the guest domain. Second, it entirely removes xenstore from the suspend/resume process. In the original code, when the migration process desired to suspend a VM it sent a message to xend, the VM management daemon. Xend in turn wrote a message to xenstore, which alerted the guest by an event channel (a virtual interrupt) that it should suspend execution. The guest's final act before suspending was to make a hypercall² which descheduled the domain and caused Xen to send a notification to xenstore, which then sent an interrupt to xend, which finally returned control to the migration process. This convoluted process could take a nearly arbitrary amount of time — typical measured latency was in the range of 30 to 40ms, but we saw delays as long as 500ms in some cases.

Remus's optimized suspend code streamlines this process by creating an event channel in the guest specifically for receiving suspend requests, which the migration process can invoke directly. Additionally, a new hypercall is provided to allow processes to register an event channel for callbacks notifying them of the completion of VM suspension. In concert, these two notification mechanisms reduce the time required to suspend a VM to about one hundred microseconds — an improvement of two orders of magnitude over the previous implementation.

In addition to these signaling changes, we have increased the efficiency of the memory copying process. First, we quickly filter out clean pages from the memory scan, because at high checkpoint frequencies most memory is unchanged between rounds. Second, we map the guest domain's entire physical memory into the replication process when it begins, rather than mapping and unmapping dirty pages at every epoch — we found that mapping foreign pages took approximately the same time as copying them.

Checkpoint support. Providing checkpoint support in Xen required two primary changes to the existing suspend-to-disk and live migration code. First, support was added for resuming execution of a domain after it had been suspended; Xen previously did not allow “live checkpoints” and instead destroyed the VM after writ-

ing its state out. Second, the suspend program was converted from a one-shot procedure into a daemon process. This allows checkpoint rounds after the first to copy only newly-dirty memory.

Supporting resumption requires two basic changes. The first is a new hypercall to mark the domain as schedulable again (Xen removes suspended domains from scheduling consideration, because previously they were always destroyed after their state had been replicated). A similar operation is necessary in order to re-arm watches in xenstore.

Asynchronous transmission. To allow the guest to resume operation as quickly as possible, the migration process was modified to copy touched pages to a staging buffer rather than delivering them directly to the network while the domain is paused. This results in a significant throughput increase: the time required for the kernel build benchmark discussed in Section 3.3 was reduced by approximately 10% at 20 checkpoints per second.

Guest modifications. As discussed above, paravirtual guests in Xen contain a suspend handler that cleans up device state upon receipt of a suspend request. In addition to the notification optimizations described earlier in this section, the suspend request handler has also been modified to reduce the amount of work done prior to suspension. In the original code, suspension entailed disconnecting all devices and unplugging all but one CPU. This work was deferred until the domain was restored on the other host. These modifications are available in Xen as of version 3.1.0.

These changes are not strictly required for correctness, but they do improve the performance of the checkpoint considerably, and involve very local modifications to the guest kernel. Total changes were under 100 lines of code in the paravirtual suspend handler. As mentioned earlier, these modifications are not necessary in the case of non-paravirtualized VMs.

2.4 Network buffering

Most networks cannot be counted on for reliable data delivery. Therefore, networked applications must either accept packet loss, duplication and reordering, or use a high-level protocol such as TCP which provides stronger service guarantees. This fact simplifies the network buffering problem considerably: transmitted packets do not require replication, since their loss will appear as a transient network failure and will not affect the correctness of the protected state. However, it is crucial that packets queued for transmission be held until the checkpointed state of the epoch in which they were generated is committed to the backup; if the primary fails, these generated packets reflect speculative state that has been lost.

Figure 3 depicts the mechanism by which we prevent the release of speculative network state. Inbound traffic is delivered to the protected host immediately, but outbound packets generated since the previous checkpoint are queued until the current state has been checkpointed and that checkpoint has been acknowledged by the backup site. We have implemented this buffer as a linux queuing discipline applied to the guest domain's network device in *domain 0*, which responds to two RT-netlink messages. Before the guest is allowed to resume execution after a checkpoint, the network buffer receives a CHECKPOINT message, which causes it to insert a barrier into the outbound queue preventing any subsequent packets from being released until a corresponding release message is received. When a guest checkpoint has been acknowledged by the backup, the buffer receives a RELEASE message, at which point it begins dequeuing traffic up to the barrier.

There are two minor wrinkles in this implementation. The first is that in linux, queuing disciplines only operate on *outgoing* traffic. Under Xen, guest network interfaces consist of a frontend device in the guest, and a corresponding backend device in *domain 0*. Outbound traffic from the guest appears as *inbound* traffic on the backend device in domain 0. Therefore in order to queue the traffic, we convert the inbound traffic to outbound by routing it through a special device called an *intermediate queuing device* [16]. This module is designed to work at the IP layer via iptables [27], but it was not difficult to extend it to work at the bridging layer we use to provide VM network access in our implementation.

The second wrinkle is due to the implementation of the Xen virtual network device. For performance, the memory used by outbound networking traffic is not copied between guest domains and domain 0, but shared. However, only a small number of pages may be shared at any one time. If messages are in transit between a guest and domain 0 for only a brief time, this limitation is not noticeable. Unfortunately, the network output buffer can result in messages being in flight for a significant amount of time, which results in the guest network device blocking after a very small amount of traffic has been sent. Therefore when queueing messages, we first copy them into local memory and then release the local mappings to shared data.

2.5 Disk buffering

Disks present a rather different challenge than network interfaces, largely because they are expected to provide much stronger reliability guarantees. In particular, when a write has been acknowledged by a disk, an application (or file system) expects to be able to recover that data even in the event of a power failure immediately fol-

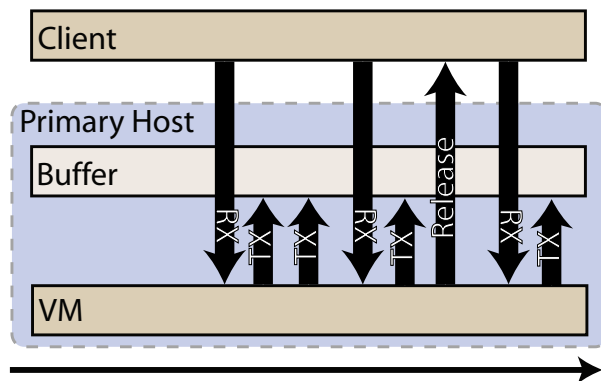


Figure 3: Network buffering in Remus.

lowing the acknowledgment. While Remus is designed to recover from a single host failure, it must preserve crash consistency even if *both* hosts fail. Moreover, the goal of providing a general-purpose system precludes the use of expensive mirrored storage hardware designed for HA applications. Therefore Remus maintains a complete mirror of the active VM's disks on the backup host. Prior to engaging the protection system, the current state of the disk on the primary is mirrored to the backup host. Once protection has been engaged, writes to persistent storage are tracked and checkpointed similarly to updates to memory. Figure 4 gives a high-level overview of the disk replication mechanism

As with the memory replication subsystem described in Section 2.3, writes to disk from the active VM are treated as write-through: they are immediately applied to the primary disk image, and asynchronously mirrored to an in-memory buffer on the backup. This approach provides two direct benefits: First, it ensures that the active disk image remains crash consistent at all times; in the case of both hosts failing, the active disk will reflect the crashed state of the externally visible VM at the time of failure (the externally visible VM resides on the primary host if the primary host has not failed or if the backup also fails before it has been activated, otherwise it resides on the backup). Second, writing directly to disk accurately accounts for the latency and throughput characteristics of the physical device. This obvious-seeming property is of considerable value: accurately characterizing disk responsiveness is a subtle problem, as we ourselves experienced in an earlier version of the disk buffer which held write requests in memory on the primary VM until checkpoint commit. Such an approach either buffers writes, under-representing the time required to commit data to disk and allowing the speculating VM to race ahead in execution, or conservatively over-estimates write latencies resulting in a loss of performance. Modeling disk access time is notoriously challenging [28], but

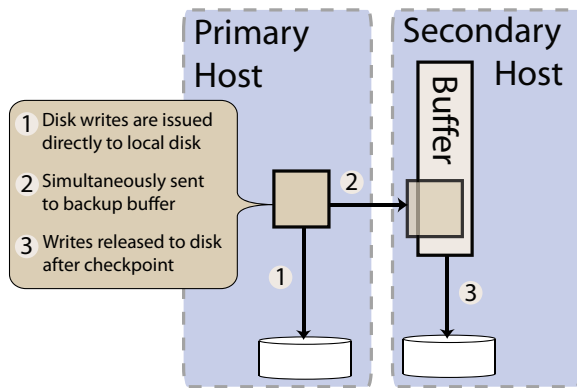


Figure 4: Disk write buffering in Remus.

our implementation avoids the problem by preserving direct feedback from the disk to its client VM.

At the time that the backup acknowledges that a checkpoint has been received, disk updates reside completely in memory. No on-disk state may be changed until the entire checkpoint has been received, as this would prevent the backup from rolling back to the most recent complete checkpoint. Once the checkpoint is acknowledged, the disk request buffer may be applied to disk. In the event of a failure, Remus will wait until all buffered writes have been applied before resuming execution. Although the backup could begin execution immediately using the request buffer as an overlay on the physical disk, this would violate the disk semantics presented to the protected VM: if the backup fails after activation but before data is completely flushed to disk, its on-disk state might not be crash consistent.

Only one of the two disk mirrors managed by Remus is actually valid at any given time. This point is critical in recovering from multi-host crashes. This property is achieved by the use of an activation record on the backup disk, which is written after the most recent disk buffer has been completely flushed to disk and *before* the backup VM begins execution. In recovering from multiple host failures, this record may be used to identify the valid, crash consistent version of the disk.

The disk buffer is implemented as a Xen block tap module [32]. The block tap is a device which allows a process in the privileged domain to efficiently interpose itself between the frontend disk device presented to a guest VM and the backend device which actually services requests. The buffer module logs disk write requests from the protected VM and mirrors them to a corresponding module on the backup, which executes the checkpoint protocol described above and then removes itself from the disk request path before the backup begins execution in the case of failure at the primary.

2.6 Detecting Failure

Remus's focus is on demonstrating that it is possible to provide advanced high availability in a general and transparent way using commodity hardware and without modifying the protected applications. We currently use a simple failure detector that is directly integrated in the checkpointing stream: a timeout of the backup responding to commit requests will result in the primary assuming that the backup has crashed and disabling protection. Similarly, a timeout of new checkpoints being transmitted from the primary will result in the backup assuming that the primary has crashed and resuming execution from the most recent checkpoint.

The system is configured to use a pair of bonded network interfaces, and the two physical hosts are connected using a pair of Ethernet crossover cables (or independent switches) on the protection NICs. Should both of these network paths fail, Remus does not currently provide mechanism to fence execution. Traditional techniques for resolving partitioning (i.e., quorum protocols) are notoriously difficult to apply in two host configurations. We feel that in this case, we have designed Remus to the edge of what is possible with commodity hardware.

3 Evaluation

Remus has been designed with the primary objective of making high availability sufficiently generic and transparent that it may be deployed on today's commodity hardware. In this section, we characterize the overheads resulting from our approach for a variety of different workloads, in order to answer two questions: (1) Is this system practically deployable? (2) What kinds of workloads are most amenable to our approach?

Before measuring the performance impact, we must establish that the system functions correctly. We accomplish this by injecting network failures at each phase of the replication protocol, while putting substantial disk, network and CPU load on the protected system. We find that the backup takes over for the lost primary within approximately one second in every case, preserving all externally visible state, including active network connections.

We then evaluate the overhead of the system on application performance across very different workloads. We find that a general-purpose task such as kernel compilation incurs approximately a 50% performance penalty when checkpointed 20 times per second, while network-dependent workloads as represented by SPECweb perform at somewhat more than one quarter native speed. The additional overhead in this case is largely due to output-commit delay on the network interface.

Based on this analysis, we conclude that although Re-

mus is efficient at state replication, it does introduce significant network delay, particularly for applications that exhibit poor locality in memory writes. Thus, applications that are very sensitive to network latency may not be well suited to this type of high availability service (although there are a number of optimizations which have the potential to noticeably reduce network delay, some of which we discuss in more detail following the benchmark results). We feel that we have been conservative in our evaluation, using benchmark-driven workloads which are significantly more intensive than would be expected in a typical virtualized system; the consolidation opportunities such an environment presents are particularly attractive because system load is variable.

3.1 Test environment

Unless otherwise stated, all tests were run on IBM eServer x306 servers, consisting of one 3.2 GHz Pentium 4 processor with hyperthreading enabled, 1 GB of RAM, 3 Intel e1000 GbE network interfaces, and an 80 GB SATA hard drive. The hypervisor was Xen 3.1.2, modified as described in Section 2.3, and the operating system for all virtual machines was linux 2.6.18 as distributed in Xen 3.1.2, with the modifications described in Section 2.3. The protected VM was allocated 512 MB of total RAM. To minimize scheduling effects from the VMM, domain 0's VCPU was pinned to the first hyperthread. One physical network interface was bridged to the guest virtual interface and used for application traffic, one was used for administrative access, and the last was used for replication (we did not bond interfaces for replication, but this is immaterial to the tests we performed). Virtual disks were provided by disk images on the SATA drive, exported to the guest using the tapdisk AIO driver.

3.2 Correctness verification

As discussed in Section 2.2, Remus's replication protocol operates in four distinct phases: (1) checkpoint changed state and increment the epoch of network and disk request streams, (2) replicate system state, (3) when the complete memory checkpoint and corresponding set of disk requests has been received, send a checkpoint acknowledgement from the backup, and (4) upon receipt of the acknowledgement, release outbound network packets queued during the previous epoch. To verify that our system functions as intended, we tested deliberately induced network failure at each stage. For each test, the protected system executed a kernel compilation process in order to generate disk, memory and CPU load. To verify the network buffer, we simultaneously executed a graphics-intensive X11 client (glxgears) attached to an external X11 server. Remus was configured to take checkpoints

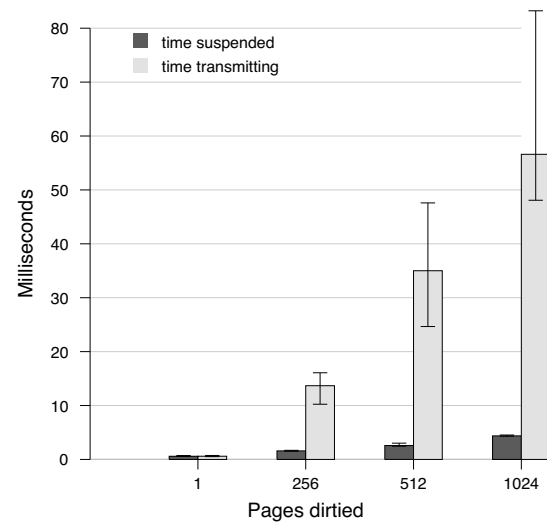


Figure 5: Checkpoint time relative to pages dirtied.

every 25 milliseconds throughout. Each individual test was repeated twice.

At every failure point, the backup successfully took over execution of the protected system, with only minor network delay (about one second) noticeable while the backup detected the failure and activated the replicated system. The glxgears client continued to run after a brief pause, and the kernel compilation task continued to successful completion. We then gracefully shut down the VM and executed a forced file system check on the backup disk image, which reported no inconsistencies.

3.3 Benchmarks

In the following section, we evaluate the performance of our system using a variety of macrobenchmarks which are meant to be representative of a range of real-world workload mixtures. The primary workloads we run are a kernel compilation test, the SPECweb2005 benchmark, and the Postmark disk benchmark. Kernel compilation is a balanced workload which stresses the virtual memory system, the disk and the CPU, SPECweb primarily exercises networking performance and memory throughput, and Postmark focuses on disk performance.

To better understand the following measurements, we performed a microbenchmark measuring the time spent copying guest state (while the guest was suspended) and the time spent sending that data to the backup relative to the number of pages changed since the previous checkpoint. We wrote an application to repeatedly change the first byte of a set number of pages and measured times over 1000 iterations. Figure 5 presents the average, minimum and maximum recorded times spent in the check-

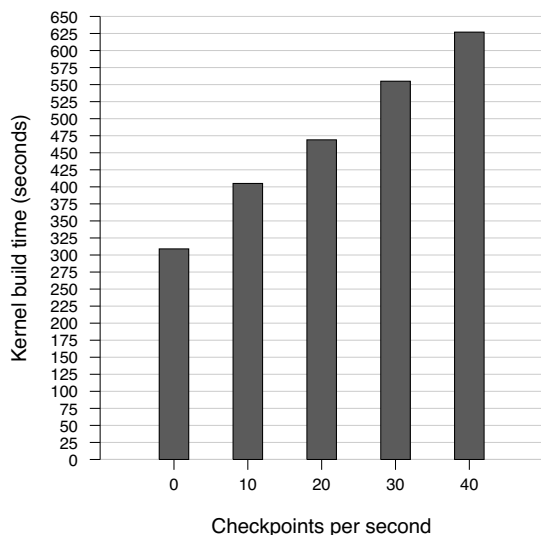


Figure 6: Kernel build time by checkpoint frequency.

point and replication stages, within a 95% confidence interval. It shows that the bottleneck for checkpoint frequency is replication time.

Kernel compilation. The kernel compile test measures the wall-clock time required to build linux kernel version 2.6.18 using the default configuration and the *bz-Image* target. Compilation uses GCC version 4.1.2, and make version 3.81. This is a balanced workload that tests CPU, memory and disk performance.

Figure 6 shows protection overhead when configured to checkpoint at rates of 10, 20, 30 and 40 times per second, compared to a baseline compilation in an unprotected virtual machine. Total measured overhead at each of these frequencies was 31%, 52%, 80% and 103%, respectively. Overhead scales linearly with checkpoint frequency within the rates we tested. We believe that the overhead measured in this set of tests is reasonable for a general-purpose system, even at a checkpoint frequency of 40 times per second.

SPECweb2005. The SPECweb benchmark is composed of at least three separate systems: a web server, an application server, and one or more web client simulators. We configure these as three VMs on distinct physical machines. The application server and the client are configured with 640 MB out of 1024 MB total available RAM. The web server and backup are provisioned with 2048 MB of RAM, of which 1024 is allocated to the web server VM, which is the system under test. The SPECweb scores we mention in this section are the highest results we achieved with the SPECweb “e-commerce” test maintaining 95% “good” and 99% “tolerable” times.

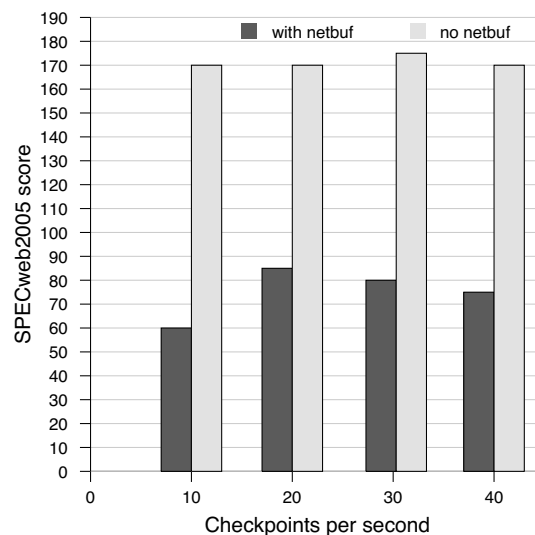


Figure 7: SPECweb scores by checkpoint frequency (native score: 305)

Figure 7 shows SPECweb performance at various checkpoint frequencies relative to an unprotected server. These scores are primarily a function of the delay imposed by the network buffer between the server and the client. Although they are configured for a range of frequencies, SPECweb touches memory rapidly enough that the time required to propagate the memory dirtied between checkpoints sometimes exceeds 100ms, regardless of checkpoint frequency. Because the network buffer cannot be released until checkpointed state has been acknowledged, the effective network delay can be higher than the configured checkpoint interval. Remus does ensure that the VM is suspended at the start of every epoch, but it cannot currently ensure that the total amount of state to be replicated per epoch does not exceed the bandwidth available during the configured epoch length. Because the *effective* checkpoint frequency is lower than the configured rate, and network latency dominates the SPECweb score, performance is relatively flat across the range of configured frequencies. At configured rates of 10, 20, 30 and 40 checkpoints per second, the average checkpoint rates achieved were 9.98, 16.38, 20.25 and 23.34 respectively, or average latencies of 100ms, 61ms, 49ms and 43ms respectively.

SPECweb is a RAM-hungry workload which is also very sensitive to network latency. This makes it a poor fit for our current implementation, which trades network delay for memory throughput. Figure 8 demonstrates the dramatic effect delay between the client VM and the web server has on SPECweb. We used the Linux *netem* [19] queueing discipline to add varying degrees of delay to

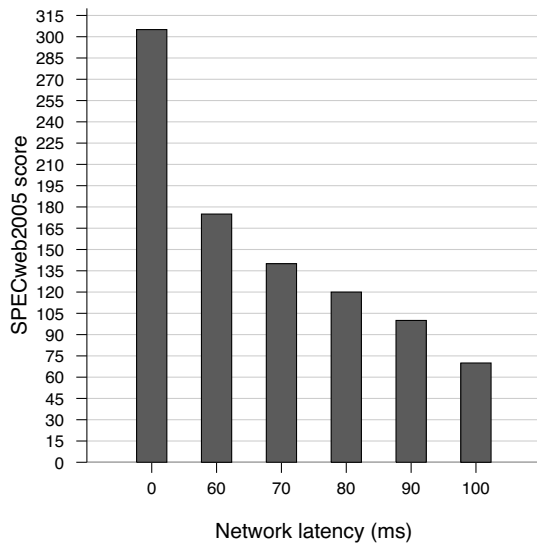


Figure 8: The effect of network delay on SPECweb performance.

the outbound link from the web server (virtualized but not running under Remus). For comparison, Figure 7 also shows protection overhead when network buffering is disabled, to better isolate network latency from other forms of checkpoint overhead (again, the flat profile is due to the effective checkpoint rate falling short of the configured rate). Deadline scheduling and page compression, discussed in Section 3.4 are two possible techniques for reducing checkpoint latency and transmission time. Either or both would reduce checkpoint latency, and therefore be likely to increase SPECweb performance considerably.

Postmark. The previous sections characterize network and memory performance under protection, but the benchmarks used put only moderate load on the disk subsystem. In order to better understand the effects of the disk buffering mechanism, we ran the *Postmark* disk benchmark (version 1.51). This benchmark is sensitive to both throughput and disk response time. To isolate the cost of disk replication, we did not engage memory or network protection during these tests. Configuration was identical to an unprotected system, with the exception that the virtual disk was provided by the tapdisk replication module. Figure 9 shows the total time required to perform 10000 postmark transactions with no disk replication, and with a replicated disk committing at frequencies of 10, 20, 30 and 40 times per second. The results indicate that replication has no significant impact on disk performance.

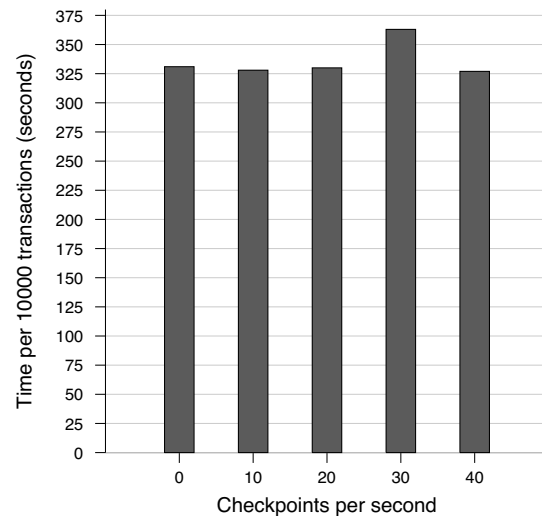


Figure 9: The effect of disk replication of Postmark performance.

3.4 Potential optimizations

Although we believe the performance overheads shown earlier in this section are reasonable for what they provide, we are eager to reduce them further, particularly for latency-sensitive workloads. In addition to more careful tuning of the existing code, we believe the following techniques have the potential to greatly increase performance.

Deadline scheduling. The amount of time required to perform a checkpoint is currently variable, depending on the amount of memory to be copied. Although Remus ensures that the protected VM is suspended at the start of each epoch, it currently makes no attempt to control the amount of state which may change between epochs. To provide stricter scheduling guarantees, the rate at which the guest operates could be deliberately slowed [10] between checkpoints, depending on the number of pages dirtied. Applications which prioritize latency over throughput, such as those modeled by the SPECweb benchmark discussed in Section 3.3, may enable this throttling for improved performance. To perform such an operation, the shadow page table handler could be extended to invoke a callback when the number of dirty pages exceeds some high water mark, or it may be configured to pause the virtual machine directly.

Page compression. It has been observed that disk writes typically only alter 5–20% of a data block [35]. If a similar property holds for RAM, we may exploit it in order to reduce the amount of state requiring replication, by sending only the delta from a previous transmission of the same page.

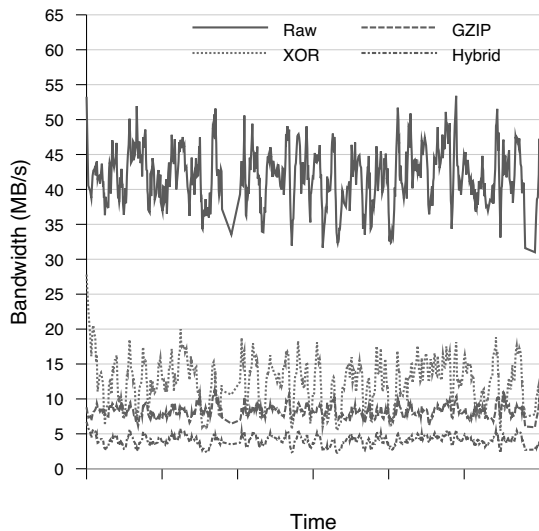


Figure 10: Comparison of bandwidth requirements using various compression schemes.

To evaluate the potential benefits of compressing the replication stream, we have prototyped a basic compression engine. Before transmitting a page, this system checks for its presence in an address-indexed LRU cache of previously transmitted pages. On a cache hit, the page is XORed with the previous version and the differences are run-length encoded. This provides significant compression when page writes do not change the majority of the page. Although this is true for much of the data stream, there remains a significant fraction of pages that have been modified to the point where XOR compression is not effective. In these cases, a general-purpose algorithm such as that used by gzip may achieve a higher degree of compression.

We found that by using a hybrid approach, in which each page is preferentially XOR-compressed, but falls back to gzip compression if the XOR compression ratio falls below 5:1 or the previous page is not present in the cache, we could observe a typical compression ratio of 10:1 on the replication stream. Figure 10 shows the bandwidth consumed in MBps for a 60-second period of the kernel compilation benchmark described in Section 3.3. The cache size was 8192 pages and the average cache hit rate was 99%.

Compressing the replication stream can consume additional memory and CPU resources on the replicating host, but lightweight schemes such as the XOR compression technique should pay for themselves through the reduction in bandwidth required for replication and consequent reduction in network buffering delay.

Copy-on-write checkpoints. The current implemen-

tation pauses the domain at each checkpoint for an amount of time linear in the number of pages which have been dirtied since the last checkpoint. This overhead could be mitigated by marking dirty pages as copy-on-write and resuming the domain immediately. This would reduce the time during which the domain must be paused to a fixed small cost proportional to the total amount of RAM available to the guest. We intend to implement copy-on-write by supplying the Xen shadow paging system with a userspace-mapped buffer into which it could copy touched pages before restoring read-write access. The replication process could then extract any pages marked as copied from the COW buffer instead of reading them directly from the guest. When it had finished replicating pages, their space in the buffer could be marked for reuse by the Xen COW module. If the buffer were to become full, the guest could simply be paused, resulting in a graceful degradation of service from COW to stop-and-copy operation.

4 Related Work

State replication may be performed at several levels, each of which balances efficiency and generality differently. At the lowest level, hardware-based replication is potentially the most robust solution. Hardware, however, is much more expensive to develop than software and thus hardware replication is at a significant economic disadvantage. Replication at the virtualization layer has many of the advantages of the hardware approach, but comes at lower cost because it is implemented in software. Like hardware, however, the virtualization layer has no semantic understanding of the operating-system and application state it replicates. As a result it can be less flexible than process checkpointing in the operating system, in application libraries or in applications themselves, because it must replicate the entire system instead of individual processes. It can also be less efficient, because it may replicate unnecessary state. The challenge for these higher-level approaches, however, is that interdependencies among state elements that comprise a checkpoint are insidiously difficult to identify and untangle from the rest of the system and thus these checkpointing mechanisms are significantly more complex than checkpointing in the virtualization layer.

Virtual machine migration. As described earlier, Remus is built on top of the Xen support for live migration [6], extended significantly to support frequent, remote checkpointing. Bradford et al. extended Xen's live migration support in another direction: migrating persistent state along with the migrating guest so that it can be restarted on a remote node that does not share network storage with the originating system[3].

Like Remus, other projects have used virtual machines

to provide high availability. The closest to our work is Bressoud and Schneider's [4]. They use the virtual machine monitor to forward the input events seen by a primary system to a backup system where they are deterministically replayed to replicate the primary's state. Deterministic replay requires much stricter constraints on the target architecture than simple virtualization and it requires an architecture-specific implementation in the VMM.

Another significant drawback of deterministic replay as exemplified by Bressoud and Schneider's work is that it does not easily extend to multi-core CPUs. The problem is that it is necessary, but difficult, to determine the order in which cores access shared memory. There have been some attempts to address this problem. For example, *Flight Data Recorder* [34] is a hardware module that sniffs cache coherency traffic in order to record the order in which multiple processors access shared memory. Similarly, Dunlap introduces a software approach in which the CREW protocol (concurrent read, exclusive write) is imposed on shared memory via page protection [8]. While these approaches do make SMP deterministic replay possible, it is not clear if they make it feasible due to their high overhead, which increases at least linearly with the degree of concurrency. Our work sidesteps this problem entirely because it does not require deterministic replay.

Virtual machine logging and replay. Virtual machine logging has been used for purposes other than high availability. For example, in *ReVirt* [9], virtualization is used to provide a secure layer for logging state changes in the target system in order to provide better forensic evidence for intrusion detection systems. The replayed system is a read-only copy of the original system, which is not meant to be run except in order to recreate the events involved in a system compromise. Logging has also been used to build a *time-travelling debugger* [13] that, like *ReVirt*, replays the system for forensics only.

Operating system replication. There are many operating systems, such as *Accent* [25], *Amoeba* [18], *MOSIX* [1] and *Sprite* [23], which support process migration, mainly for load balancing. The main challenge with using process migration for failure recovery is that migrated processes typically leave residual dependencies to the system from which they were migrated. Eliminating these dependencies is necessary to tolerate the failure of the primary host, but the solution is elusive due to the complexity of the system and the structure of these dependencies.

Some attempts have been made to replicate applications at the operating system level. *Zap* [22] attempts to introduce a virtualization layer within the linux kernel. This approach must be rebuilt for every operating system, and carefully maintained across versions.

Library approaches. Some application libraries provide support for process migration and checkpointing. This support is commonly for parallel application frameworks such as *CoCheck* [29]. Typically process migration is used for load balancing and checkpointing is used to recover an entire distributed application in the event of failure.

Replicated storage. There has also been a large amount of work on checkpointable storage for disaster recovery as well as forensics. The Linux Logical Volume Manager [14] provides a limited form of copy-on-write snapshots of a block store. *Parallax* [33] significantly improves on this design by providing limitless lightweight copy-on-write snapshots at the block level. The *Andrew File System* [11] allows one snapshot at a time to exist for a given volume. Other approaches include *RSnapshot*, which runs on top of a file system to create snapshots via a series of hardlinks, and a wide variety of backup software. *DRBD* [26] is a software abstraction over a block device which transparently replicates it to another server.

Speculative execution. Using speculative execution to isolate I/O processing from computation has been explored by other systems. In particular, *SpecNFS* [20] and *Rethink the Sync* [21] use speculation in a manner similar to us in order to make I/O processing asynchronous. *Remus* is different from these systems in that the semantics of block I/O from the guest remain entirely unchanged: they are applied immediately to the local physical disk. Instead, our system buffers generated network traffic to isolate the externally visible effects of speculative execution until the associated state has been completely replicated.

5 Future work

This section briefly discusses a number of directions that we intend to explore in order to improve and extend *Remus*. As we have demonstrated in the previous section, the overhead imposed by our high availability service is not unreasonable. However, the implementation described in this paper is quite young. Several potential areas of optimization remain to be explored. Upon completion of the targeted optimizations discussed in Section 3.4, we intend to investigate more general extensions such as those described below.

Introspection optimizations. *Remus* currently propagates more state than is strictly necessary. For example, buffer cache pages do not need to be replicated, since they can simply be read in from persistent storage on the backup. To leverage this, the virtual disk device could log the addresses of buffers provided to it for disk reads, along with the associated disk addresses. The memory-copying process could then skip over these pages if they

had not been modified after the completion of the disk read. The remote end would be responsible for reissuing the reads from its copy of the disk in order to fill in the missing pages. For disk-heavy workloads, this should result in a substantial reduction in state propagation time.

Hardware virtualization support. Due to the lack of equipment supporting hardware virtualization in our laboratory at the time of development, we have only implemented support for paravirtualized guest virtual machines. But we have examined the code required to support fully virtualized environments, and the outlook is quite promising. In fact, it may be somewhat simpler than the paravirtual implementation due to the better encapsulation provided by virtualization-aware hardware.

Cluster replication. It would be useful to extend the system to protect multiple interconnected hosts. While each host can be protected independently, coordinated protection would make it possible for internal network communication to proceed without buffering. This has the potential to dramatically improve the throughput of distributed applications, including the three-tiered web application configuration prevalent in managed hosting environments. Support for cluster replication could be provided by a distributed checkpointing protocol such as that which is described in our colleague Gang Peng's master's thesis [24], which used an early version of the checkpointing infrastructure provided by Remus.

Disaster recovery. Remus is a product of the Second-Site [7] project, whose aim was to provide geographically diverse mirrors of running systems in order to survive physical disaster. We are in the process of planning a multi-site deployment of Remus in order to experiment with this sort of configuration. In a long distance deployment, network delay will be an even larger concern. Additionally, network reconfigurations will be required to redirect Internet traffic accordingly.

Log-structured datacenters. We are extending Remus to capture and preserve the complete execution history of protected VMs, rather than just the most recent checkpoint. By mapping guest memory into Parallax [17], our virtual block store designed for high-frequency snapshots, we hope to be able to efficiently store large amounts of both persistent and transient state at very fine granularity. This data should be very useful in building advanced debugging and forensics tools. It may also provide a convenient mechanism for recovering from state corruption whether introduced by operator error or by malicious agents (viruses and so forth).

6 Conclusion

Remus is a novel system for retrofitting high availability onto existing software running on commodity hardware. The system uses virtualization to encapsulate a protected

VM, and performs frequent whole-system checkpoints to asynchronously replicate the state of a single speculatively executing virtual machine.

Providing high availability is a challenging task and one that has traditionally required considerable cost and engineering effort. Remus commodifies high availability by presenting it as a service at the virtualization platform layer: HA may simply be "switched on" for specific virtual machines. As with any HA system, protection does not come without a cost: The network buffering required to ensure consistent replication imposes a performance overhead on applications that require very low latency. Administrators must also deploy additional hardware, which may be used in N-to-1 configurations with a single backup protecting a number of active hosts. In exchange for this overhead, Remus completely eliminates the task of modifying individual applications in order to provide HA facilities, and it does so without requiring special-purpose hardware.

Remus represents a previously unexplored point in the design space of HA for modern servers. The system allows protection to be simply and dynamically provided to running VMs at the push of a button. We feel that this model is particularly attractive for hosting providers, who desire to offer differentiated services to customers.

Acknowledgments

The authors would like to thank their paper shepherd, Arun Venkataramani, and the anonymous reviewers for their insightful and encouraging feedback. We are also indebted to Anoop Karollil for his aid in the evaluation process. This work is supported by grants from Intel Research and the National Science and Engineering Research Council of Canada.

References

- [1] BARAK, A., AND WHEELER, R. Mosix: an integrated multiprocessor unix. 41–53.
- [2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 164–177.
- [3] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM Press, pp. 169–179.
- [4] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault-tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (December 1995), pp. 1–11.
- [5] CHANDRA, S., AND CHEN, P. M. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *IS-SRE '02: Proceedings of the 13th International Symposium on*

- Software Reliability Engineering (ISSRE'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 91.
- [6] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association.
 - [7] CULLY, B., AND WARFIELD, A. Secondsite: disaster protection for the common server. In *HOTDEP'06: Proceedings of the 2nd conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2006), USENIX Association.
 - [8] DUNLAP, G. *Execution Replay for Intrusion Analysis*. PhD thesis, University of Michigan, 2006.
 - [9] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)* (2002).
 - [10] GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. To infinity and beyond: time warped network emulation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (2005).
 - [11] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* 6, 1 (1988), 51–81.
 - [12] HP. NonStop Computing. <http://h20223.www2.hp.com/non-stopcomputing/cache/76385-0-0-0-121.aspx>.
 - [13] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association.
 - [14] Lvm2. <http://sources.redhat.com/lvm2/>.
 - [15] MARQUES, D., BRONEVETSKY, G., FERNANDES, R., PINGALI, K., AND STODGHILL, P. Optimizing checkpoint sizes in the c3 system. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)* (April 2005).
 - [16] MCHARDY, P. Linux imq. <http://www.linuximq.net/>.
 - [17] MEYER, D., AGGARWAL, G., CULLY, B., LEFEBVRE, G., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Parallax: Virtual disks for virtual machines. In *EuroSys '08: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM.
 - [18] MULLENDER, S. J., VAN ROSSUM, G., TANENBAUM, A. S., VAN RENESSE, R., AND VAN STAVEREN, H. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53.
 - [19] netem. <http://linux-net.osdl.org/index.php/Netem>.
 - [20] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 191–205.
 - [21] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association.
 - [22] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.* 36, SI (2002), 361–376.
 - [23] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The sprite network operating system. *Computer* 21, 2 (1988), 23–36.
 - [24] PENG, G. Distributed checkpointing. Master's thesis, University of British Columbia, 2007.
 - [25] RASHID, R. F., AND ROBERTSON, G. G. Accent: A communication oriented network operating system kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles* (New York, NY, USA, 1981), ACM Press, pp. 64–75.
 - [26] REISNER, P., AND ELLENBERG, L. Drbd v8 – replicated storage with shared disk semantics. In *Proceedings of the 12th International Linux System Technology Conference* (October 2005).
 - [27] RUSSELL, R. Netfilter. <http://www.netfilter.org/>.
 - [28] SCHINDLER, J., AND GANGER, G. Automated disk drive characterization. Tech. Rep. CMU SCS Technical Report CMU-CS-99-176, Carnegie Mellon University, December 1999.
 - [29] STELLNER, G. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)* (Honolulu, Hawaii, 1996).
 - [30] SYMANTEC CORPORATION. Veritas Cluster Server for VMware ESX. http://eval.symantec.com/mktginfo/products/Datasheets/High_Availability/vcs22vmware_datasheet.pdf, 2006.
 - [31] VMWARE, INC. VMware high availability (ha). <http://www.vmware.com/products/vi/vc/ha.html>, 2007.
 - [32] WARFIELD, A. *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, 2006.
 - [33] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: managing storage for a million machines. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association.
 - [34] XU, M., BODIK, R., AND HILL, M. D. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ACM Press, pp. 122–135.
 - [35] YANG, Q., XIAO, W., AND REN, J. Trap-array: A disk array architecture providing timely recovery to any point-in-time. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 289–301.

Notes

¹Remus buffers network and disk output. Other devices, such as the console or the serial port, are presumed to be used for local administration and therefore would not require buffering. However, nothing prevents these devices from being buffered as well.

²Paravirtual Xen guests contain code specifically for suspend requests that are responsible for cleaning up Xen-related state such as shared memory mappings used by virtual devices. In the case of hardware virtualized (e.g., Windows) VMs, this state is completely encapsulated by Xen's device model, and these in-guest changes are unnecessary.

Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures*

Chi Ho, Robbert van Renesse
Cornell University

Mark Bickford
ATC-NY

Danny Dolev
Hebrew University of Jerusalem

Abstract

The paper presents and evaluates *Nysiad*,¹ a system that implements a new technique for transforming a scalable distributed system or network protocol tolerant only of crash failures into one that tolerates arbitrary failures, including such failures as freeloading and malicious attacks. The technique assigns to each host a certain number of *guard hosts*, optionally chosen from the available collection of hosts, and assumes that no more than a configurable number of guards of a host are faulty. Nysiad then enforces that a host either follows the system's protocol and handles all its inputs fairly, or ceases to produce output messages altogether—a behavior that the system tolerates. We have applied Nysiad to a link-based routing protocol and an overlay multicast protocol, and present measurements of running the resulting protocols on a simulated network.

1 Introduction

Scalable distributed systems have to tolerate nondeterministic failures from causes such as Heisenbugs and Mandelbugs [14], aging related or bit errors (*e.g.*, [23]), selfish behavior (*e.g.*, freeloading), and intrusion. While all these failures are prevalent and it would seem that large distributed systems have sufficient redundancy and diversity to handle such failures, developing software that delivers scalable Byzantine fault tolerance has proved difficult, and few such systems have been built and deployed. Distributed systems and protocols like DNS, BGP, OSPF, IS-IS, as well as most P2P communication systems tolerate only crash failures. Secure versions of such systems aim to prevent compromise of par-

ticipants. While important, this issue is orthogonal to tolerating Byzantine failures as a host is not faulty until it is compromised.

We know how to build practical scalable Byzantine-tolerant data stores (*e.g.*, [1, 2, 27]). Various work has also focused on Byzantine-tolerant peer-to-peer protocols (*e.g.*, [3, 9, 20, 17, 19]). However, the only known and general approach to developing a Byzantine version of a protocol or distributed system is to replace each host by a Replicated State Machine (RSM) [18, 25]. As replicas of a host can be assigned to existing hosts in the system, this does not necessarily require a large investment of hardware.

This paper presents Nysiad, a technique that uses a variant of RSM to make distributed systems tolerant of Byzantine failures in asynchronous environments, and evaluates the practicality of the approach. Nysiad leverages that most distributed systems already deal with crash failures and, rather than masking arbitrary failures, translates arbitrary failures into crash failures. Doing so avoids having to solve consensus [12] during normal operation. Nysiad invokes consensus only when a host needs to communicate with new peers or when one of its replicas is being replaced.

Instead of treating replicas as symmetric, Nysiad's replication scheme is essentially primary-backup with the host that is being replicated acting as primary. Different from RSM's original specification [18], Nysiad allows the entire RSM to halt in case the host does not comply with the protocol. A voting protocol ensures that the output of the RSM is valid. A credit-based flow control protocol ensures that the RSM processes all its inputs (including external input) fairly. As a result of combining both properties, the worst that the Byzantine host can accomplish is to stop processing input, a behavior that the original system will treat as a crash failure and recover accordingly.

We believe that the cost of Nysiad, while significant, is within the range of practicality for mission-critical ap-

*The authors were supported by AFRL award FA8750-06-2-0060 (CASTOR), NSF award 0424422 (TRUST), AFOSR award FA9550-06-1-0244 (AF-TRUST), DHS award 2006-CS-001-000001 (I3P), as well as by ISF, ISOC, CCR, and Intel Corporation. The views and conclusions herein are those of the authors.

¹The Nysiads nursed Dionysos and rendered him immortal.

plications. End-to-end message latencies grow by a factor of 3 compared to message latencies in the original system. The overhead caused by public key cryptography operations are manageable. Most alarmingly, the total number of messages sent in the translated system per end-to-end message sent in the original system can grow significantly depending on factors such as the communication behavior of the original system. However, the message overhead does not grow significantly as a function of the total number of hosts, and grows only linearly as a function of the number of failures to be tolerated. Most of the additional traffic is in the form of control messages that do not carry payload.

The paper

- evaluates for the first time the overheads involved when applying RSM to scalable distributed systems;
- shows that RSM does not require solving consensus if the original system is already tolerant of crash failures;
- presents a novel technique that forces hosts to process input fairly in a Byzantine environment, or leave;
- demonstrates how automatic reconfiguration can be supported in a Byzantine-tolerant distributed system.

Section 2 presents background and related work on countering Byzantine behavior. Section 3 describes an execution model and introduces terminology. The Nysiad design is presented in Section 4. Section 5 provides notes on the implementation. Section 6 evaluates the performance of systems generated by Nysiad using various case studies. Limitations are discussed in Section 7. Section 8 concludes.

2 Background

The RSM approach can be applied to systems like DNS [7]. While overheads are practical, the approach does not handle reconfiguration in the DNS hierarchy.

The idea of automatically translating crash-tolerant systems into Byzantine systems can be traced back to the mid-eighties. Gabriel Bracha presents a translation mechanism that transforms a protocol tolerant of up to t crash failures into one that tolerates t Byzantine failures [6]. Brian Coan also presents a translation [11] that is similar to Bracha's. These approaches have two important restrictions. One is that input protocols are required to have a specific style of execution, and in particular they have to be round-based with each participant awaiting the receipt of $n - t$ messages before starting a new

round. Second, the approaches have quadratic message overheads and as a result do not scale well. Note that these approaches were primarily intended for a certain class of consensus protocols, while we are pursuing arbitrary protocols and distributed systems.

Toueg, Neiger and Bazzi worked on an extension of Bracha's and Coan's approaches for translation of synchronous systems [4, 5, 24]. Mpoeleng et al. [22] present a scalable translation that is also intended for synchronous systems, and transforms Byzantine faults to so-called *signal-on-failure* faults. They replace each host with a pair, and assume only one of the hosts in each pair may fail. In the Internet, making synchrony assumptions is dangerous. Byzantine hosts can easily trigger violations of such assumptions to attack the system.

Closely related to Nysiad is the recent PeerReview system [15], providing accountability [28] in distributed systems. PeerReview detects those Byzantine failures that are observable by a correct host. Like Nysiad, PeerReview assumes that each host implements a protocol using a deterministic state machine. PeerReview maintains incoming and outgoing message logs and, periodically, runs incoming logs through the state machines and checks output against outgoing logs. PeerReview can only detect a subclass of Byzantine failures, and only after the fact. Like reputation management and intrusion detection systems, accountability deters intentionally faulty behavior, but does not prevent or tolerate it.

Nysiad is based on our prior work described in [16], in which we developed a theoretical basis for a similar translation technique, but one that does not scale, does not handle reconfiguration, and does not prevent a Byzantine host from considering its input selectively.

3 Model

A *system* is a collection of hosts that exchange messages as specified by a *protocol*. Below we will use the terms *original* and *translated* to refer to the systems before and after translation, respectively. The original system tolerates only crash failures, while the translated system tolerates Byzantine failures as well. For simplicity we will assume that each host runs a deterministic state machine that transitions in response to receiving messages or expiring timers. (Nysiad may handle nondeterministic state machines by considering nondeterministic events as inputs.) As a result of input processing, a state machine may *produce* messages, intended to be sent to other hosts. The system is assumed to be asynchronous, with no bounds on event processing, message latencies, or clock drift.

The hosts are configured in an undirected *communication graph* (V, E) , where V is the set of hosts and E the set of communication links. A host only communi-

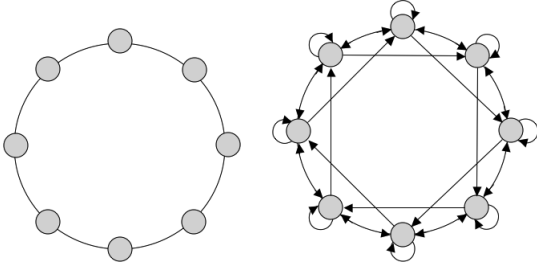


Figure 1: A communication graph (left) and a possible guard graph (right) for $t = 1$. In this particular case, each host has exactly $3t + 1$ guards, and each set of neighbors exactly $2t + 1$ monitors.

communicates directly with its adjacent hosts, also called *neighbors*. The graph may change over time, for example as hosts join and leave the system. We will initially assume that the graph is static and known to all hosts. We later weaken this assumption and address topology changes.

The Nysiad transformation requires that the communication graph has a *guard graph*. A t -guard graph of (V, E) is a directed graph (V', E') with the following requirements:

1. Each host in V has a (directed) edge to at least $3t + 1$ hosts in V' (including itself) called the *guards* of the host.
2. For each two neighboring hosts in V , the two hosts have edges to at least $2t + 1$ common guards in V' (including themselves). We call such guards the *monitors* of the two hosts.

We assume that for each host in V at most t of its guards are Byzantine. We also assume that messages between correct guards of the same host are eventually delivered (using an underlying retransmission protocol that recovers from message loss). Note that a monitor of two hosts is a guard of both hosts, and that neighbors in V are each other's guards. Moreover, each host is one of its own guards.

Within the constraints of these requirements, Nysiad works with any guard graph. For efficiency it is important to create as few guards per host as possible, as all guards of a host need to be kept synchronized. However, the requirements on guards and monitors may produce guard graphs with some of the hosts needing more than $3t + 1$ guards.

If $V = V'$, no additional hosts are added to the system and hosts guard one another. Figure 1 presents an example communication graph and a possible guard graph for $t = 1$ where no additional hosts were added. Some deployments may favor adding additional hosts for the sole purpose of guarding hosts in the original system.

In the current implementation of Nysiad, the guard graph is created and maintained by a logically centralized, Byzantine-tolerant service called the *Olympus*, described in Section 4.4. The Olympus certifies the guards of a host, and is involved only when the communication graph changes as a result of host churn or new communication patterns in the original system. The Olympus does not need to be aware of the protocol that the original system employs.

4 Design

Nysiad translates the original system by replicating the deterministic state machine of a host onto its guards. Nysiad is composed of four subprotocols. The *replication protocol* ensures that guards of a host remain synchronized. The *attestation protocol* guarantees that messages delivered to guards are messages produced by a valid execution of the protocol. The *credit protocol* forces a host to either process all its input fairly, or to ignore all input. Finally, the *epoch protocol* allows the guard graph to be bootstrapped and reconfigured in response to host churn. The following subsections describe each of these protocols. The final subsection describes how Nysiad deals with external I/O.

4.1 Replication

The state machine of a host is replicated onto the guards of the host, together constituting a Replicated State Machine (RSM). It is important to keep in mind that we replicate a host only for ensuring integrity, not for availability or performance reasons. After all, the original system can already maintain availability in the face of unavailable hosts.

Say α_j^i is the replica of the state machine of host h_i on guard h_j . A host h_i broadcasts input events for its local state machine replica α_i^i to its guards. A guard h_j delivers an input event to α_j^i when h_j receives such a broadcast message from h_i . In order to guarantee that the guards of h_i stay synchronized in the face of Byzantine behavior, the hosts use a reliable ordered broadcast protocol called *OARcast* (named for Ordered Authenticated Reliable Broadcast) [16] for communication.

Using OARcast a host can *send* a message that is intended for all its guards. When a guard host h_j *delivers a message m from h_i* it means that h_j received m , believes it came from h_i , and delivers m to α_j^i , the replica of h_i 's state machine on guard h_j . OARcast guarantees the following properties:

- *Relay*. If h_j and h_k are correct, and h_j delivers m from h_i , then h_k delivers m from h_i (even if h_i is Byzantine);

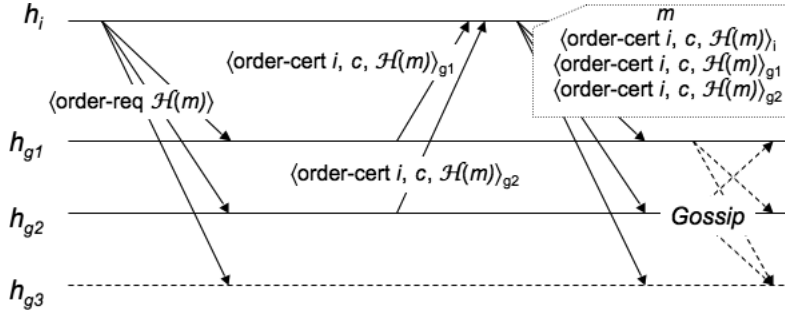


Figure 2: Host h_i initiates an OARcast execution for $t = 1$. The time diagram shows all guards of h_i , where only h_{g3} is faulty.

- *Ordering.* If two hosts h_j and h_k are correct and h_j and h_k both deliver m_1 from h_i and m_2 from h_i , then they do so in the same order (even if h_i is Byzantine);
- *Authenticity.* If two hosts h_i and h_j are correct and h_i does not send m , then h_j does not deliver m from h_i ;
- *Persistence.* If two hosts h_i and h_j are correct, and h_i sends a message m , then h_j delivers m from h_i ;
- *FIFO.* If two hosts h_i and h_j are correct, and h_i sends a message m_1 before m_2 , then h_j delivers m_1 from h_i before delivering m_2 from h_i .

Relay guarantees that all correct guards deliver a message if one correct guard does. *Ordering* guarantees that all correct guards deliver messages from the same host in the same order. These two properties together guarantee that the correct replicas of a host stay synchronized, even if the host is Byzantine. *Authenticity* guarantees that Byzantine hosts cannot forge messages of correct hosts. *Persistence* rules out a trivial implementation that does not deliver any messages. *FIFO* stipulates that correct guards deliver messages from a correct host in the order sent.

These properties are not as strong as those for asynchronous consensus [12] and indeed consensus is not necessary for our use, as only the host whose state is replicated can issue updates (*i.e.*, there is only one *proposer*). If that host crashes or stops producing updates for some other reason, no new host has to be elected to take over its role, and the entire RSM is allowed to halt as a result. Indeed, unlike consensus, the OARcast properties can be realized in an asynchronous environment with failures, as we shall show next.

The implementation of OARcast used in this paper is as follows. Say a sending host $h_i \in V$ wants to OARcast an input message m to its n_i guards in V' ,

where ($n_i > 3t$). Each guard h_j maintains a sequence number c on behalf of h_i . Using private (MAC-authenticated) FIFO point-to-point connections, h_i sends $\langle \text{order-req } \mathcal{H}(m) \rangle$ to each guard, where \mathcal{H} is a cryptographic one-way hash function. On receipt, h_j sends an *order certificate* $\langle \text{order-cert } i, c, \mathcal{H}(m) \rangle_j$ back to h_i , where the subscript j means that h_j digitally signed the message such that any host can check its origin.

As at most t of h_i 's guards are Byzantine, h_i is guaranteed to receive *order-cert* messages from at least $n_i - t$ different guards with the correct sequence number and message hash. We call such a collection of *order-cert* messages an *order proof for* (c, m) . Byzantine orderers cannot generate conflicting order proofs (same sequence number, different messages) even if h_i itself is Byzantine, as two order proofs have at least $t+1$ *order-cert* messages in common, one of which is guaranteed to be generated by a correct guard [21].

h_i delivers m locally to α_i^i and forwards m along with an order proof to each of its guards. On receipt, a guard h_j checks that the order proof corresponds to m and is for the next message from h_i . If so, h_j delivers m to α_j^i .

To guarantee the *Relay* property, h_j gossips order proofs with the other guards. A similar implementation of OARcast is proved correct in [16]. That paper also presents an implementation that does not use public key cryptography, but has higher message overhead.

Figure 2 shows an example of an OARcast execution. Optimizations are discussed in Section 5. Not counting the overhead of gossip and without exploiting hardware multicast, a single OARcast to $3t$ guards uses at most $9t$ messages. Gossip can be largely piggybacked on existing traffic.

4.2 Attestation

While the replication protocol above guarantees that guards of a host synchronize on its state, it does not guarantee that the host OARcasts *valid* input events, because

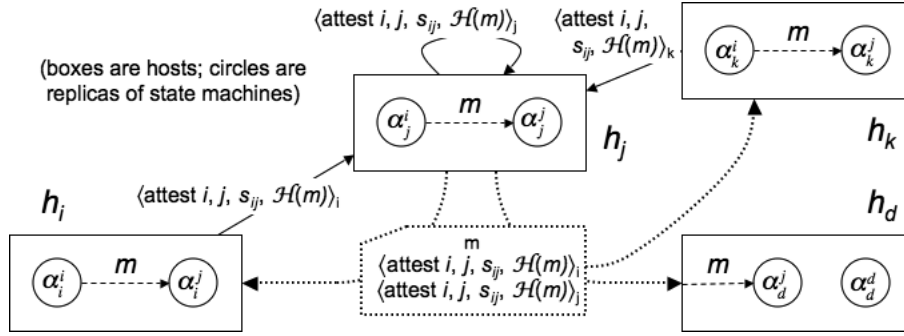


Figure 3: Normal case attestation when $t = 1$. Here the state machine of h_i sends a message m to the state machine of h_j . The guards of h_j are h_i , h_k , h_d , and h_j itself, and each run a replica of h_j 's state machine. Hosts h_i , h_j , and h_k monitor h_i and h_j . h_j collects attestations for m and OARcasts the event to its guards. In this case only h_d needs the attestations.

the sending host h_i may forge arbitrary input events. We consider two kinds of input events: message events and timer events. Checking validity for each is slightly different.

First let us examine message sending. Say in the original system host $h_i \in V$ sends a message m to a host $h_j \in V$. Because h_j is a neighbor of h_i it is also a guard of h_i , and thus in the translated system α_j^j will produce m as an input event for α_j^j . Accordingly h_j OARcasts m to its guards, but the guards, not sure whether to trust h_j , need a way to verify the validity of m before delivering m to local replicas. To protect against Byzantine behavior of h_j , we require that h_j includes a proof of validity with every OARcast in the form of a collection of $t + 1$ attestations from guards of h_i .

Because the guards of h_i implement an RSM, each (correct) guard $h_k \in V'$ has a replica α_k^i of the state of h_i that produces m . Each guard h_k of h_i (including h_i and h_j) sends $\langle \text{attest } i, j, s_{ij}, \mathcal{H}(m) \rangle_k$ to h_j . s_{ij} is a sequence number for messages from i to j , and prevents replay attacks. h_j has to collect t of these attestations in addition to its own and include them in its OARcast to convince h_j 's guards of the validity of m . Again, correct guards have to gossip attestations in order to guarantee that every correct guard receives them in case one does.

There are two important optimizations to this. First, as h_j only needs $t + 1$ attestations, only the monitors of h_i and h_j need to send attestations to guarantee that h_j gets enough attestations. This not only reduces traffic, but the monitors are neighbors of h_j and thus no additional communication links need be created. Second, h_j does not need the attestations until the last phase of the OARcast protocol, thus h_j can request order certificates before it has received the attestations. This way ordering and attestation can happen in parallel rather than sequentially. Both these optimizations are exploited in the im-

plementation (Section 5). Figure 3 shows an example of the flow of traffic when using attestations.

In case of a timer event at a host h , h needs to collect t additional attestations of its own guards in addition to its own attestation. This prevents h from producing timer events at a rate higher than that of the fastest correct host. While theoretically this may appear useless in an asynchronous environment, in practice doing so is important. Consider, for example, a system in which a host pings its neighbors in order to verify that they are alive. Without timer attestation, a Byzantine host may force a failure detection by not waiting long enough for the response to those pings. While in an asynchronous system one cannot detect failures accurately using a ping-pong protocol, timer attestation ensures in this case that a host has to wait a reasonable amount of time. Also, because hosts only wait for t attestations from more than $2t$ guards, Byzantine guards cannot delay or block timer events emitted by correct hosts.

4.3 Credits

While attestation prevents a host from forging invalid input events, a host may still selectively ignore input events and fail to produce certain output events. For example, in the ping-pong example above, a host could respond to pings, avoiding failure detection, but neglect to process other events. In a multicast tree application a host could accept input but neglect to forward output to its children (freeloading). Such a host could even deny wrongdoing by claiming that it has not yet received the input events or that the output events have already been sent but simply not yet delivered by the network—after all, we assume that the system is asynchronous.

We present a credit-based approach that forces hosts either to process input from all sources fairly and pro-

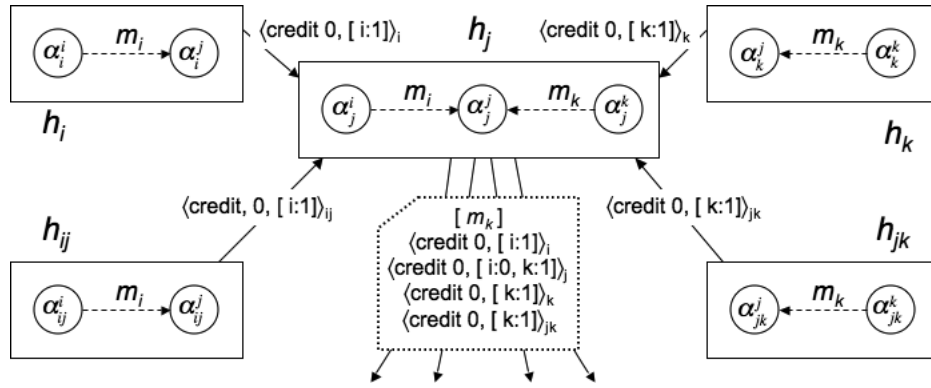


Figure 4: Credit mechanism with $t = 1$. h_i and h_k are neighbors of h_j , each sending a message to h_j . h_j tries to order the message from h_k while ignoring the message from h_i . The credit mechanism renders the OARcast illegal.

duce the corresponding output events, or to cease processing altogether. The essence of the idea is to require that a host obtain credits from its guards in order to OARcast new input events, and a guard only complies if it has received the OARcast from the host for previous input events. As such, credits are the flip-side of attestations: while attestations prevent a host from producing bad output, credits force a host to either process all input or process none of it. If a host elects to process no input, it cannot produce output and will eventually be considered as a crashed host by the original system.

We will exploit that a single OARcast from a host can order a sequence of pending input events for its state machine, rather than one input message at a time. The OARcast's order certificate binds a single sequence number to the ordered list of input events. We say that the OARcast *orders* the events in the list.

A credit is a signed object of the form $\langle \text{credit } j, c, \vec{v}_{i,j} \rangle_i$, where h_i has to be a guard of h_j . h_i sends such a credit to h_j to approve delivery of the c^{th} OARcast message from h_j , provided a certain ordering condition specified by $\vec{v}_{i,j}$ holds. Including c prevents replay attacks. The ordering constraint $\vec{v}_{i,j}$ is a vector that contains an entry for each state machine that h_i and h_j both guard. Such an entry contains how many events (possibly 0) the corresponding state machine replica on h_i has produced for α_i^j .

For each neighbor h_k of h_j , h_j has to collect at least $t + 1$ credits for OARcast c from monitors of h_j and h_k . However, h_j can only use a credit for an OARcast if the OARcast orders all messages specified in the credit's ordering constraint that were not ordered already by OARcasts numbered less than c . These two constraints taken together guarantee that an OARcast contains a credit from a correct monitor for each of its neighbors, and prevents h_j from ignoring input messages that correct monitors observe while ordering other input messages.

For example, consider Figure 4, showing five hosts. h_i and h_k are neighbors of h_j . h_{ij} is a monitor for hosts h_i and h_j , while h_{jk} is a monitor for h_j and h_k . Assume $t = 1$. Consider a situation in which h_j has not yet sent any OARcasts, but α_i^i has produced a message m_i for h_j on hosts h_i and h_{ij} , while α_k^k has produced a message m_k for h_j on hosts h_k and h_{jk} . Each guard of h_j sends credit for the first OARcast that reflects the messages produced locally for h_j .

Now assume that h_j is Byzantine and trying to ignore messages from h_i but process messages from h_k . h_j has to include a credit from either h_i or h_{ij} . Because h_j is Byzantine and $t = 1$, both h_i and h_{ij} have to be correct and will not collude with h_j . If h_j tries to order only m_k as shown in the figure, receiving hosts will note that at least one credit requires that a message from h_i has to be ordered and will therefore ignore the OARcast (and report the message to authorities as proof of wrongdoing).

As with other credit-based flow control mechanisms, a *window* w may be used to allow for pipelining of messages. Initially, each guard of h_j sends credits for the first w OARcasts from h_j , specifying an empty ordering constraint. Then, on receipt of the c^{th} OARcast, a guard sends a credit for OARcast $c + w$, using an ordering constraint that reflects the current set of produced messages for h_j . If $w = 1$, the next OARcast cannot be issued until it has been received by at least $t + 1$ monitors of each neighbor and the new credits have been communicated to h_j . If $w > 1$ pipelining becomes possible, but at the expense of additional freedom for h_j . In practice we found that $w = 2$ enables good performance while monitors still have significant control over the order of messages produced by the hosts they guard.

4.4 Epochs

So far we have assumed that the communication graph (V, E) and its t -guard graph (V', E') are static and well-known to all hosts. This is necessary, because when a host receives an OARcast it has to check that the order certificates, the attestations, and the credits were generated by qualified hosts. In particular, order certificates and credits have to be generated by a guard of the sending host of an OARcast message, and each attestation of a message has to be generated by monitors of the source and destination of the message. Also, the receiving host of an OARcast has to know how many guards the sending host has in order to check that a message contains a sufficient number of ordering certificates and credits.

While Nysiad, in theory, could inspect the code of the state machines, it has no good way of determining which hosts will be communicating with which other hosts, and so in reality even the communication graph (V, E) is initially unknown, let alone its guard graph. Making matters worse, such a communication graph often evolves over time.

In order to handle this problem, we introduce a logically centralized (but Byzantine-replicated [10]) trusted certification service that we call the *Olympus*. The Olympus is not involved in normal communication, but only in charge of tracking the communication graph and updating the guard graph accordingly. The Olympus produces signed *epoch certificates* for hosts, which contain sufficient information for a receiver of an OARcast message to check its validity. In particular, an epoch certificate for a host h_i describes

- the host identifier (i);
- the set of the identifiers of all h_i 's guards;
- the *epoch number* (described below);
- a hash of the final state of the host in the previous epoch.

The Olympus does not need to know the protocol that the original system uses. Initially, the Olympus assigns $3t$ guards to each host arbitrarily, in addition to the host itself. Each guard starts in epoch 0 and runs the state machine starting from a well-defined initial state. Order certificates and credits have to contain the epoch number in order to prevent replay attacks of old certificates in later epochs. Next we describe a general protocol for changing guards and how this protocol is used to handle reconfigurations.

Changing-of-the-guards

While the Olympus assigns guards to hosts, the *changing-of-the-guards* protocol starts with the guards

themselves. In response to triggers that we will describe below, each guard of h_i sends a *state certificate* containing the current epoch number and a secure hash of its current state to h_i . After the guard receives an acknowledgment from h_i it is free to clean up its replica, unless the guard is h_i itself. However, in order to avoid replay attacks the guard needs to remember that this epoch of h_i 's execution has terminated.

When h_i has received $n_i - t$ such certificates (typically including its own) that correspond to its own state, h_i sends the collection of state certificates to the Olympus. $n_i - t$ certificates together guarantee that there are at most t correct guards and t Byzantine guards that are still active, not enough to order additional OARcast messages. Effectively, the collection certifies that the state machine of h_i has halted in the given state.

In response, the Olympus assigns new guards to h_i and creates a new epoch certificate using an incremented epoch number and the state hash, and sends the certificate to h_i . On receipt, h_i sends its signed state and the new epoch certificate to its new collection of guards. Recipients check validity of the state using the hash in the epoch certificate and resume normal operation.

Reconfiguration

One scenario in which the changing-of-the-guards protocol is triggered is when guards of h_i produce a message m for another host h_j for the first time. Each correct guard sends a state certificate to h_i when it produces the message. *The state has to be such that the message m is about to be produced, so that when the state machine is later restarted, possibly on a different guard, m is produced and processed the normal way.* The state certificate also indicates that a message for h_j is being produced, so that the Olympus may know the reason for the invocation.

h_i collects $n_i - t$ state certificates, and sends the collection to the Olympus. The Olympus, now convinced that h_i has produced a message for h_j , requests h_j to change its guards as well. h_j does this by OARcasting a special end-epoch message, triggering the changing-of-the-guards protocol at each guard in the same state. (Should h_j not respond then it is up to the Olympus to decide when to declare h_j faulty, block h_j 's guards, and restart h_i .)

Assuming the Olympus has received the state certificates for both h_i and h_j , the Olympus can assign new guards to each in order to satisfy the constraints of the guard graph. The Olympus then sends new epoch certificates to both h_i and h_j , after which each sends its certificate to its new guards. These guards start in a state where they first produce m , which can now be processed normally.

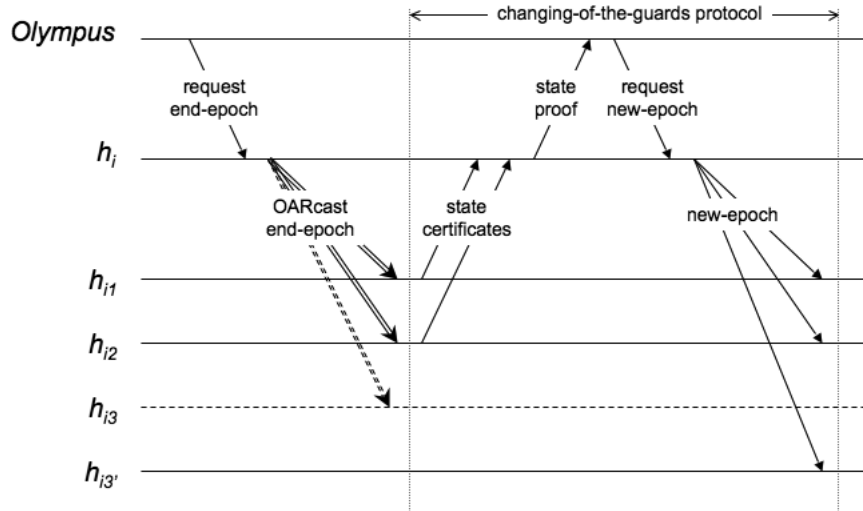


Figure 5: Example of an execution of the reconfiguration protocol. h_{i1} , h_{i2} , and h_{i3} are guards of h_i . When the Olympus suspects that h_{i3} has failed, it requests the current epoch of h_i to be concluded and installs a new set of guards, replacing h_{i3} with $h_{i3'}$.

The Olympus also undertakes reconfiguration when it determines that a guard of a host has failed. In order to detect crash failures, the Olympus may periodically ping all guards to determine responsiveness. (A more scalable solution is described in [17]. Note that while a false positive may introduce overhead, it is not a correctness issue.) Also, guards send proof of observable Byzantine behavior to the Olympus. In response to detection of a failure of a guard of a host other than the host itself, the Olympus requests the host to OARcast an *end-epoch* message to invoke the changing-of-the-guards protocol. Figure 5 shows an example.

Should a host $h_i \in V$ be detected as faulty then the Olympus sends a message to all h_i 's guards, requesting them to block further OARcasts from h_i . Once the Olympus has received acknowledgments from $n_i - t$ guards, the Olympus knows that h_i can no longer produce input for other hosts successfully.

4.5 External I/O

So far we have assumed that Nysiad translates a system in its entirety. However, often such a system serves external clients that cannot easily be treated in the same way. We cannot expect to be able to replicate those clients onto multiple hosts, and it becomes impossible to verify that the clients send valid data using a general technique. To wit, a Byzantine-tolerant storage service does not verify the validity of the data that it stores, nor does a Byzantine-tolerant multicast service check the data from the broadcaster. The usual assumption, from the system's point of view, is to *trust* clients.

In Nysiad, we treat external clients as trusted hosts. Such hosts may crash or leave, but there is no need to replicate their state machines, nor to attest the data they generate. However, when a trusted host h_i sends a message to an untrusted host h_j , we do want to make sure that h_j treats the input fairly with respect to other inputs that it receives. Vice versa, when h_j sends a message to h_i , h_i has to collect attestations in order to verify the validity of the message. We also want to prevent h_j from withholding messages for h_i .

The methodology we developed so far can be adapted to achieve these requirements. We assign the pair (h_i, h_j) $2t + 1$ *half-monitors*. Each half-monitor runs a full replica of h_j 's state machine, but for h_i only keeps track of the messages that h_i sends. Unlike normal monitors, h_i itself does not run a half-monitor, but h_j does.

When h_i wants to send a message to h_j , it sends a copy of the message to each half-monitor using authenticated FIFO channels. (The half-monitors gossip the receipt of this message with one another to ensure that either all or none of the correct half-monitors receive the message in a situation in which h_i crashes during sending.) Like normal monitors, half-monitors generate attestations for messages from h_i so that h_j can convince others of the validity of that input. More importantly, half-monitors generate credits for h_j forcing h_j to treat h_i 's messages fairly with respect to its other inputs.

In a similar manner, half-monitors generate attestations for messages from h_j to h_i so that h_i can verify the validity of those messages. Should h_j itself fail to

send messages to h_i then the half-monitors can provide the necessary copy.

5 Implementation Details

In order to evaluate overheads we implemented Nysiad in Java. In this section we provide details on how we construct guard graphs and how we combine the various subprotocols into a single coherent protocol.

Given a communication graph and a parameter t many different guard graphs are often possible. For efficiency and fault tolerance it is prudent to minimize the number of guards per host (see Section 3). We are not aware of an optimal algorithm for determining such a graph. We devised the following algorithm to create a t -guard graph of the communication graph. It runs in two phases. In the first phase the algorithm considers each pair of neighbors (h_i, h_j) . Initially h_i and h_j are assigned as monitors. The algorithm then determines the hosts that are 1 hop away from the current set of monitors, and adds, randomly, such hosts to the set of monitors until there are no such hosts left or until the number of monitors has reached $2t+1$. This step is repeated until the set of monitors has reached the required size. Note that the monitors are guards to both h_i and h_j . In the next phase, the algorithm considers all hosts individually. If a host has fewer than $3t+1$ guards then the closest hosts in terms of hop distance are added, randomly as before, until the desired number of guards is reached.

While best understood separately, the OARcast, attestation, and credit protocols combine into a single replication protocol. Doing so reduces message and CPU overheads significantly, while also simplifying implementation. Consider the c^{th} OARcast from some host h_i , and assume h_i has the necessary credits and has produced the messages required by those credits. At this point h_i creates an `order-req`, containing a list of hashes of the messages that it has produced but not yet ordered in previous OARcasts, and sends the request to each of its n_i guards.

On receipt, each guard signs a *single* certificate that contains the credit for OARcast $c+w$, an order certificate for OARcast c , and any attestations that it can create for messages in OARcast c . This way the signing and checking costs of all certificate types can be amortized. The guard sends the resulting certificate back to h_i . h_i awaits $n_i - t$ certificates, which collectively are guaranteed to contain the necessary order certificates and attestations for completing the current OARcast, and the necessary credits for OARcast $c+w$.

In the third and final round, h_i sends these aggregate certificates to its guards. On receipt, a guard has to check the signatures on all certificates except its own. The end-to-end latency consists of three network latencies, plus

the latency of signing (done in parallel by each of the guards) and checking $n_i - 1$ certificates (executed in parallel as well). The more messages can be ordered by a single OARcast, the more these costs can be amortized.

An execution of OARcast requires $3 \cdot (n_i - 1)$ FIFO messages. Since $n_i > 3t$, the minimum number of FIFO messages per OARcast is $9t$. In order to further reduce traffic, Nysiad also tries to combine messages for different OARcasts—if two FIFO messages are sent at approximately the same time between two different hosts, they are combined in a manner similar to back-to-back messages in the TCP protocol.

6 Case Studies

While one cannot test if a system tolerates Byzantine failures, it is possible to measure the overheads involved. In this section we report on two case studies: a point-to-point link-level routing protocol and a peer-to-peer multicast protocol. We applied Nysiad to each and ran the result over a simulated network to measure network overheads and overheads caused by cryptographic operations.

For the point-to-point routing protocol we selected Scalable Source Routing (SSR) [13]. SSR is inspired by the Chord overlay routing protocol [26], but can be deployed on top of the link layer. (SSR is similar to Virtual Ring Routing [8], which applies the same idea to Pastry.)

The basic idea of SSR is simple. Each host initially knows its own (location-independent) identifier and those of the neighbors it is directly connected to. The SSR protocol organizes the hosts into a Chord-like ring by having each host discover a source route to its successor and predecessor. This is done as follows. Initially a host h_i sends a message to its best guess at its successor. Should this tentative successor host know of a better successor for h_i , or discover one later, then the successor host sends a source route for the better successor back to h_i . On receipt h_i sends a message to its new best guess at its successor, and so on. This protocol converges into the desired ring and terminates. Once the ring is established routing can be done in a Chord-like manner, whereby a message travels around the ring, but taking shortcuts whenever possible. In our simulations we measure the ring-discovery protocol, not the routing itself.

The multicast protocol is even simpler. Here we assume that the hosts are organized in a balanced binary tree, and that each host forwards messages from its parent to its children (if any). We call this protocol MCAST. We measured the overhead of sending a message from the root host to all hosts.

We considered two network graph configurations. In the first, Tree, the network graph is a balanced binary tree. In the second, Random, we placed hosts uniformly

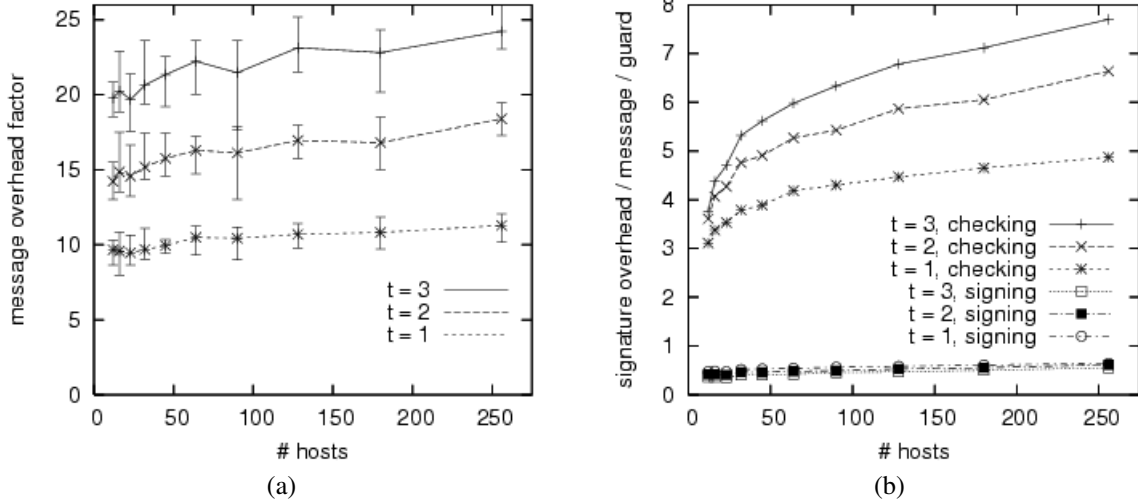


Figure 6: Message overhead factor (a) and public key signing and checking overheads (b) as a function of the number of hosts for running SSR on a Random graph using $k = 3$ and various t .

at random on a square metric space, and connected each host to its k closest peers.

We report on three configurations:

- **SSR/Random** The SSR protocol on top of a Random graph;
- **SSR/Tree** The SSR protocol on a Tree graph;
- **MCAST/Tree** The MCAST protocol on a Tree graph.

For the evaluation we developed a simple discrete time event network simulator to evaluate message overheads. The fidelity of the simulation was kept low in order to scale the simulation experiments to interesting sizes. While the simulator models network latency, we assume bandwidth is infinite. The public key signature operations were replaced by simple hash functions. We focus our evaluation on the failure-free “normal case” executions. We vary the number of hosts and t , and in the case of the Random graph we also vary k , the (minimum) number of neighbors of each host. In all experiments, the credits window w was chosen to be 2.

By and large, the increase in latency is close to a factor of 3 for all experiments, independent of what parameters are chosen. (No graphs shown.) This amount of increase was expected as the OARcast protocol consists of three rounds of communication (see Section 5). This can be decreased to two rounds by having the guards broadcast certificates directly to each other, but this results in a message overhead that is quadratic in t rather than linear.

When measuring message overhead, we report on the ratio between the number of FIFO messages sent in the

translated protocol and the number of FIFO messages sent in the original protocol. We call this the *message overhead factor*, and report the minimum, average, and maximum over 10 executions. We ignore messages sent on behalf of the gossip protocol that implement the Relay property of OARcast. These messages do not require additional cryptographic operations and contribute only a small and constant load on the network.

For measuring CPU overhead, we report only the number of public key signing and checking operations per message per guard. Such operations tend to dominate protocol processing overheads. We found the variance for these measurements to be low, the minimum and maximum usually being within 1 operation from the average number of operations, and so we report only the averages.

In the first set of experiments, we used the SSR/Random configuration using a Random graph with $k = 3$. In Figure 6(a) we show the message overhead factor for $t = 1, 2, 3$. As we described in Section 5, an OARcast to n guards uses at most $3n$ messages, and we see that this explains the trends well. There is an increase in overhead as we increase the number of hosts due to an increase in the average number of guards per host and reduced opportunity for aggregation as traffic becomes less concentrated due to the larger graph. Small graphs necessitate more sharing of guards, which reduces overhead.

Figure 6(b) reports, per guard the average number of public key sign and check operations per message in the original system. Due to aggregation, the number of sign operations message in the original system per guard is

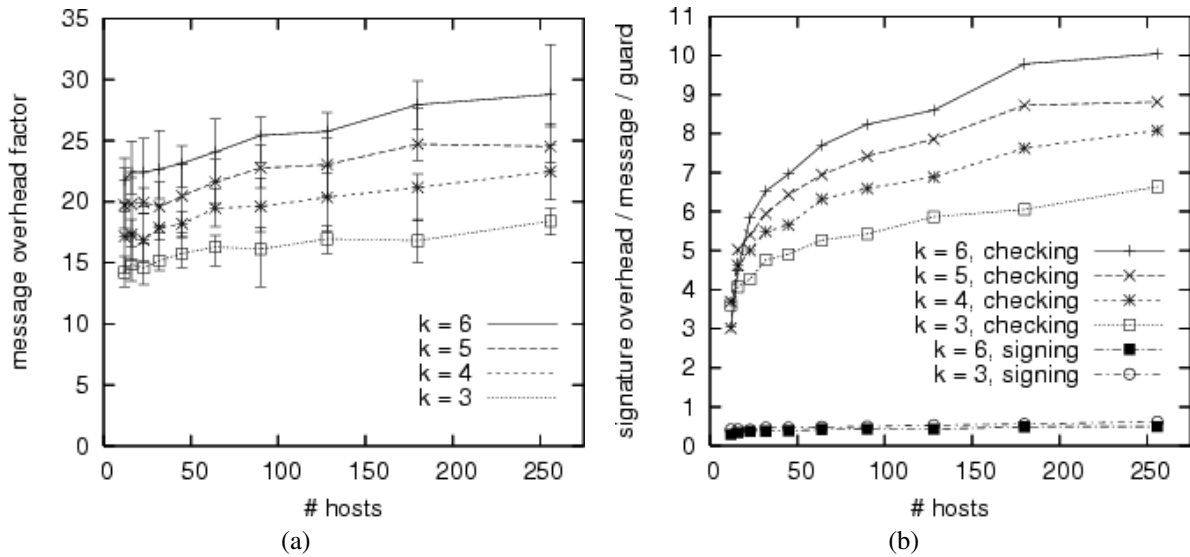


Figure 7: Message overhead (a) and public key signing and checking overheads (b) as a function of the number of hosts for the SSR protocol on a Random graph using $t = 2$ and various k , the minimum number of neighbors per host.

always less than 1 and does not significantly depend on t , as can be understood from Section 5. However, guards have to check each other's signatures and The number of check operations per message per guard may exceed $3t$ because a host may have more than $3t + 1$ guards, and, as stated above larger graphs tend to have more guards. Nonetheless, these graphs should also reach an asymptote.

Next, for the same SSR/Random configuration, we fix $t = 2$ and range k from 3 to 6. We show the message and public key signature overhead measurements in Figure 7. Even though t is fixed, an increase in the number of neighbors per host requires additional monitors, and thus the average number of guards per host tends to increase beyond the required $3t + 1$, causing additional message and CPU overhead. It is thus important for overhead of translation and indeed for fault tolerance to configure the original protocol to use as sparse a graph as possible. This tends to increase the diameter of the communication graph, and thus a suitable trade-off has to be designed.

In the final experiments, we compare the three different configurations for $t = 1$. For the Random graph we chose $k = 3$. In the case of a Tree graph, the average number of neighbors per host is approximately 2, internal hosts having 3 neighbors, leaf hosts having 1 neighbor, and the root host having 2 neighbors. We report results in Figure 8.

MCAST suffers most message overhead. This is because there is no opportunity for message aggregation in the experiment—each host receives only one message

(from its parent). However, when multiple messages are streamed, the opportunity for message aggregation is excellent—any backlog that builds up can be combined and ordered using a single OARcast operation—and thus throughput is not limited by this overhead. Even if messages cannot be aggregated, order certificates, attestations, and credits still can, and thus signature generation and checking overheads are still good.

SSR performs significantly better on the Tree graph than on the Random graph. Because communication opportunities are more limited in the Tree graph with fewer neighbors to choose from, many messages can be aggregated and ordered simultaneously. For such situations the message overhead can indeed completely disappear.

Finally, note that if hardware multicast were available the overhead of Nysiad could be significantly reduced (from $9t$ point-to-point messages for an OARcast in the best case to $3t$ point-to-point messages and 2 multicasts).

7 Discussion

Nysiad can generate a Byzantine-tolerant version of a system that was designed to tolerate only crash failures. This comes with significant overheads. When developing a Byzantine-tolerant file system, such overheads are easily masked by the overhead of accessing the disk and large data transfers. When applied to message routing protocols where there is no disk overhead and payload sizes are relatively small, overheads cannot be masked as easily.

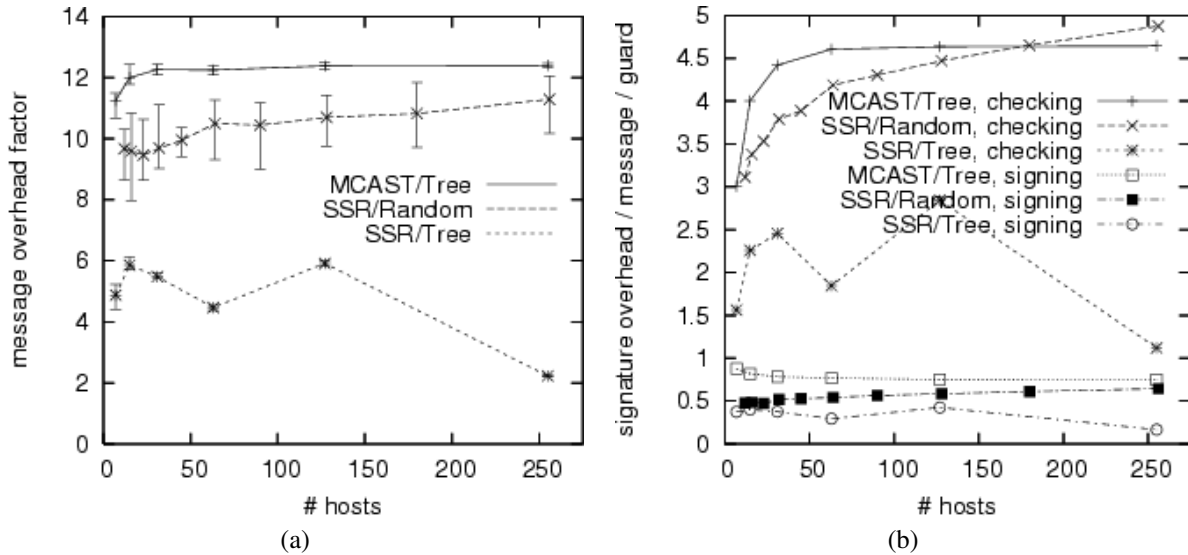


Figure 8: Message overhead factor (a) and public key signing and checking overheads (b) as a function of the number of hosts for various protocols and graphs using $t = 1$ and $k = 3$.

In practice, Nysiad may be used to generate a *first cut* at a Byzantine-tolerant protocol or distributed system, and then apply application-specific optimizations that maintain correctness. For example, if it is possible to distinguish the retransmission of a data packet from the original transmission, then it may be possible for the original transmission to be routed unguarded. Doing so could potentially mask most overhead of Nysiad.

But even if such optimizations are not possible, some applications may choose robustness over raw speed. Byzantine fault tolerance can be a part of increasing security, but it does not solve all security problems. Nysiad is not intended to defend against intrusion, but to tolerate intrusions. Defense against intrusion involves authentication and authorization techniques, as well as intrusion detection, and these are essential to guarantee that there is sufficient diversity among guards and no more than a small fraction are compromised. In the face of a limited number of successful intrusions Nysiad maintains integrity and availability of a system, but it does not provide confidentiality of data. Worse still, the replication of state complicates confidentiality. Hosts cannot trust their guards for confidentiality, and confidential data has to be encrypted in an end-to-end fashion.

Another possibility is to run some of the mechanisms that Nysiad uses inside secured hosts that are more difficult to compromise than hosts “in the field.” Such secured hosts may have reduced general functionality and use their resources to guard a relatively large number of state machines.

Nysiad makes strong assumptions about how many hosts can fail using the threshold value t . But what happens if more than t guards of a host become Byzantine? Now the host can in fact behave in a Byzantine fashion and break the system. As a system becomes larger it becomes more likely that a host has more than t Byzantine guards, and thus t should be chosen large enough to handle the maximum system size. If N is the maximum system size, then t should be chosen $O(\log N)$ in order to keep the probability that any host in the system has more than t Byzantine guards sufficiently low. As [17] demonstrates, a value for t of 2 or 3 is probably sufficient for most applications. It is also important that, as much as possible, proofs of observed Byzantine behavior are sent to the Olympus immediately so that faulty hosts can be removed quickly [28].

Nysiad exploits diversity and is defenseless against deterministic bugs that either cause a host to make an incorrect state transition or allow an attacker to compromise more than t host. The use of configuration wizards, high-level languages, and bug-finding tools may help avoid such problems. Similarly, Nysiad is helpless in the face of link-level Denial-of-Service attacks. These should be controlled by network-level anti-DoS techniques.

Nysiad in its current form uses the Olympus, a logically centralized service, to handle configuration changes. Because the Olympus is not invoked during normal operation, the load on the Olympus is likely sufficiently low for many practical applications. This architecture does not deal well with high churn, nor does the translated protocol handle network partitions well: hosts

that cannot communicate with the Olympus are excluded from participating.

Finally, we have evaluated the use of Nysiad for systems where each host has a relatively small number of neighbors with which it communicates actively. Figure 7 shows that overhead grows as a function of the number of neighbors. In systems where hosts have many active neighbors the overhead of the Nysiad protocols could be substantial. We are considering a variant of Nysiad where not all neighbors of a host are guards in order to contain overhead.

8 Conclusion

Nysiad is a general technique for developing scalable Byzantine-tolerant systems and protocols in an asynchronous environment that does not require consensus to be solved. Starting with a system tolerant of crash failures only, Nysiad assigns a set of guards to each host that verify the output of the host and constrain the order in which the host handles its inputs. A logically centralized service assigns guards to hosts in response to churn in the communication graph. Simulation results show that Nysiad may be practical for a large class of distributed systems.

References

- [1] ADYA, A., BOLOSKY, W., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX.
- [2] AMIR, Y., DANILOV, C., KIRSCH, J., LANE, J., DOLEV, D., NITA-ROTARU, C., OLSEN, J., AND ZAGE, D. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 06)* (Washington, DC, June 2006).
- [3] AWERBUCH, B., AND SCHEIDELER, C. Group Spreading: A protocol for provably secure distributed name service. In *Proc. of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)* (Turku, Finland, July 2004), vol. 3142 of *Lecture Notes in Computer Science*, Springer.
- [4] BAZZI, R. *Automatically increasing fault tolerance in distributed systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1995.
- [5] BAZZI, R., AND NEIGER, G. Simplifying fault-tolerance: providing the abstraction of crash failures. *J. ACM* 48, 3 (2001), 499–554.
- [6] BRACHA, G. Asynchronous Byzantine agreement protocols. *Inf. Comput.* 75, 2 (1987), 130–143.
- [7] CACHIN, C., AND SAMAR, A. Secure distributed DNS. In *Proc. of the Int. Conf. on Dependable Systems and Networks DSN 04* (Florence, Italy, June 2004).
- [8] CAESAR, M., CASTRO, M., NIGHTINGALE, E., O’SHEA, G., AND ROWSTRON, A. Virtual Ring Routing: Network routing inspired by DHTs. In *Proc. of SIGCOMM’06* (Pisa, Italy, Sept. 2006).
- [9] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the 5th Usenix Symposium on Operating System Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).
- [10] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)* (New Orleans, LA, Feb. 1999).
- [11] COAN, B. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers* 37, 12 (1988), 1541–1553.
- [12] FISCHER, M., LYNCH, N., AND PATTERSON, M. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (Apr. 1985), 374–382.
- [13] FUHRMANN, T. The use of Scalable Source Routing for networked sensors. In *Proc. of the 2nd IEEE Workshop on Embedded Networked Sensors* (Sydney, Australia, May 2005), pp. 163–165.
- [14] GROTTKE, M., AND TRIVEDI, K. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer* 40, 2 (Feb. 2007).
- [15] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *Proc. of the 21st ACM Symp. on Operating Systems Principles* (Stevenson, WA, Oct. 2007).
- [16] HO, C., DOLEV, D., AND VAN RENESSE, R. Making distributed systems robust. In *Proc. of the 11th Int. Conf. on Principles Of Distributed Systems (OPODIS’07)* (Guadeloupe, French West Indies, Dec. 2007).

- [17] JOHANSEN, H., ALLAVENA, A., AND VAN RENESSE, R. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. of Eurosys 2006* (Leuven, Belgium, Apr. 2006).
- [18] LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. *Trans. on Programming Languages and Systems* 6, 2 (Apr. 1984), 254–280.
- [19] LI, H., CLEMENT, A., WONG, E., NAPPER, J., ROY, I., ALVISI, L., AND DAHLIN, M. BAR gossip. In *Proc. of the 2006 USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)* (Nov. 2006).
- [20] M., H., AND VAN RENESSE, R. Defense against intrusion in a live streaming multicast system. In *Proc. of the Sixth IEEE International Conference on Peer-to-Peer Computing (P2P '06)* (Washington, DC, Sept. 2006), IEEE Computer Society.
- [21] MALKHI, D., AND REITER, M. Byzantine Quorum Systems. *Distributed Computing* 11 (June 1998), 203–213.
- [22] MPOELENG, D., EZHILCHELVAN, P., AND SPEIRS, N. From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 03)* (Los Alamitos, CA, 2003), IEEE Computer Society.
- [23] MUKHERJEE, S., EMER, J., AND REINHARDT, S. The soft error problem: An architectural perspective. In *Proc. of the Symposium on High-Performance Computer Architecture* (Feb. 2005).
- [24] NEIGER, G., AND TOUEG, S. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing* (Toronto, Ontario, Aug. 1988), ACM SIGOPS-SIGACT.
- [25] SCHNEIDER, F. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22, 4 (Dec. 1990), 299–319.
- [26] STOICA, I., MORRIS, R., KARGER, D., AND KAASHOEK, M. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the '95 Symp. on Communications Architectures & Protocols* (Cambridge, MA, Aug. 1995), ACM SIGCOMM.
- [27] WEATHERSPOON, H., EATON, P., CHUN, B.-G., AND KUBIATOWICZ, J. Antiquity: exploiting a secure log for wide-area distributed storage. In *Proc. of the 2007 EuroSys Conf.* (Lisbon, Portugal, 2007).
- [28] YEMEREFENDI, A., AND CHASE, J. The role of accountability in dependable distributed systems. In *Proc. of the First Workshop on Hot Topics in System Dependability (HotDep'05)* (Yokohama, Japan, June 2005), IEEE.

BFT Protocols Under Fire

Atul Singh^{†‡} Tathagata Das^{*} Petros Maniatis^φ Peter Druschel[†] Timothy Roscoe^x
[†]MPI-SWS, [‡]Rice University, ^{*}IIT Kharagpur, ^φIntel Research Berkeley, ^xETH Zürich

Abstract

Much recent work on Byzantine state machine replication focuses on protocols with improved performance under benign conditions (LANs, homogeneous replicas, limited crash faults), with relatively little evaluation under typical, practical conditions (WAN delays, packet loss, transient disconnection, shared resources). This makes it difficult for system designers to choose the appropriate protocol for a real target deployment. Moreover, most protocol implementations differ in their choice of runtime environment, crypto library, and transport, hindering direct protocol comparisons even under similar conditions.

We present a simulation environment for such protocols that combines a declarative networking system with a robust network simulator. Protocols can be rapidly implemented from pseudocode in the high-level declarative language of the former, while network conditions and (measured) costs of communication packages and crypto primitives can be plugged into the latter. We show that the resulting simulator faithfully predicts the performance of native protocol implementations, both as published and as measured in our local network.

We use the simulator to compare representative protocols under identical conditions and rapidly explore the effects of changes in the costs of crypto operations, workloads, network conditions and faults. For example, we show that Zyzzyva outperforms protocols like PBFT and Q/U under most but not all conditions, indicating that one-size-fits-all protocols may be hard if not impossible to design in practice.

1 Introduction

Byzantine Fault-Tolerant (BFT) protocols for replicated systems have received considerable attention in the systems research community [3, 7, 9], for applications including replicated file systems [6], backup [4], and block

stores [10]. Such systems are progressively becoming more mature, as evidenced by recent designs sufficiently fine-tuned and optimized to approach the performance of centralized [14] or crash-fault only [10] systems in some settings.

Much of the attraction of such systems stems from the combination of a simple programming interface with provable correctness properties under a strong adversarial model. All a programmer need do is write her server application as a sequential, misbehavior-oblivious state machine; available BFT protocols can replicate such application state machines across a population of replica servers, guaranteeing *safety* and *liveness* even in the face of a bounded number of arbitrarily faulty (Byzantine) replicas among them. The safety property (*linearizability*) ensures that requests are executed sequentially under a single schedule consistent with the order seen by clients. The liveness property ensures that all requests from correct clients are eventually executed.

Though these protocols carefully address such correctness properties, their authors spend less time and effort evaluating BFT protocols under severe—yet benign—failures. In fact, they often optimize under the assumption that such failures do not occur. For example, Zyzzyva [14] obtains a great performance boost under the assumption that all replica servers have good, predictable latency¹ to their clients, whereas Q/U [3] significantly improves its performance over its precursors assuming no service object is being updated by more than one client at a time.

Unfortunately, even in the absence of malice, deviations from expected behavior can wreak havoc with complex protocols. As an example from the non-Byzantine world, Junqueira et al. [11] have shown that though the “fast” version of Paxos consensus² operates in fewer rounds than the “classic” version of Paxos (presumably resulting in lower request latency), it is nevertheless more vulnerable to variability in replica connectivity. Because fast Paxos requires more replicas (two-thirds of

the population) to participate in a round, it is as slow as the slowest of the fastest two-thirds of the population; in contrast, classic Paxos is only as slow as the median of the replicas. As a result, under particularly skewed replica connectivity distributions, the two rounds of fast Paxos can be slower than the three rounds of classic Paxos. This is the flavor of understanding we seek in this paper for BFT protocols. We wish to shed light on the behavior of BFT replication protocols under adverse, yet benign, conditions that do not affect correctness, but may affect tangible performance metrics such as latency, throughput, and configuration stability.

As we approach this objective, we rely on simulation. We present BFTSim, a simulation framework that couples a high-level protocol specification language and execution system based on P2 [18] with a computation-aware network simulator built atop ns-2 [1] (Section 3). P2's declarative networking language (OverLog) allows us to capture the salient points of each protocol without drowning in the details of particular thread packages, cryptographic primitive implementations, and messaging modules. ns-2's network simulation enables us to explore a range of network conditions that typical testbeds cannot easily address. Using this platform, we implemented from scratch three protocols: the original PBFT [6], Q/U [3], and Zyzyva [14]. We validate our simulated protocols against published results under corresponding network conditions. Though welcome, this is surprising, given that all three systems depend on different types of runtime libraries and thread packages, and leads us to suspect that a protocol's performance characteristics are primarily inherent in its high-level design, not the particulars of its implementation.

Armed with our simulator, we make an “apples to apples” comparison of several BFT protocols under identical conditions. Then, we expose the protocols to benign conditions that push them outside their comfort zone (and outside the parameter space typically exercised in the literature), but well within the realm of possibility in real-world deployment scenarios. Specifically, we explore latency and bandwidth heterogeneity between clients and replicas, and among replicas themselves, packet loss, and timeout misconfiguration (Section 4). Our primary goal is to test conventional (or published) wisdom with regards to which protocol or protocol type is better than which; it is rare that “one size fits all” in any engineering discipline, so understanding the envelope of network conditions under which a clear winner emerges can be invaluable.

While we have only begun to explore the potential of our methodology, our study has already led to some interesting discoveries. Among those, perhaps the broadest statement we can make is that though agreement protocols offer hands down the best throughput,

quorum-based protocols tend to offer lower latency in wide-area settings. Zyzyva, the current state-of-the-art agreement-based protocol provides almost universally the best throughput in our experiments, except in a few cases. First, Zyzyva is dependent on timeout settings at its clients that are closely tied to client-replica latencies; when those latencies are not uniform, Zyzyva tends to fall back to behavior similar to a two-phase quorum protocol like HQ [9], as long as there is no write contention. Second, with large request sizes, Zyzyva's throughput drops and falls slightly below Q/U's and PBFT's with batching, since its primary is required to send full requests to all the backup replicas. Lastly, under high loss rates, Zyzyva tends to compensate quickly and expensively, causing its response time to exceed that of the more mellow Q/U.

Section 2 provides some background on BFT replicated state machines. In Section 3, we explain our experimental methodology, describe our simulation environment, and validate it by comparing its predictions with published performance results on several existing BFT protocols we have implemented in BFTSim. Section 4 presents results of a comparative evaluation of BFT protocols under a wide range of conditions. We discuss related works in Section 5 and close with future work and conclusions in Section 6.

2 Background

In this section, we discuss the work on which this paper is based: BFT replicated state machines. Specifically, we outline the basic machinery of the protocols we study in the rest of this paper: PBFT by Castro and Liskov [6], Q/U by Abd-El Malek et al. [3], and Zyzyva and Zyzyva5 by Kotla et al. [14].

At a high level, all such protocols share the basic objective of assigning each client request a unique order in the global service history, and executing it in that order. Agreement-based protocols such as PBFT first have the replicas communicate with each other to agree on the sequence number of a new request and, when agreed, execute that request after they have executed all preceding requests in that order. PBFT has a three-phase agreement protocol among replicas before it executes a request. Quorum protocols, like Q/U, instead restrict their communication to be only between clients and replicas—as opposed to among replicas; each replica assigns a sequence number to a request and executes it as long as the submitting client appears to have a current picture of the whole replica population, otherwise uses conflict resolution to bring enough replicas up to speed. Q/U has a one-phase protocol in the fault-free case, but when faults occur or clients contend to write the same object the protocol has more phases. Zyzyva is a

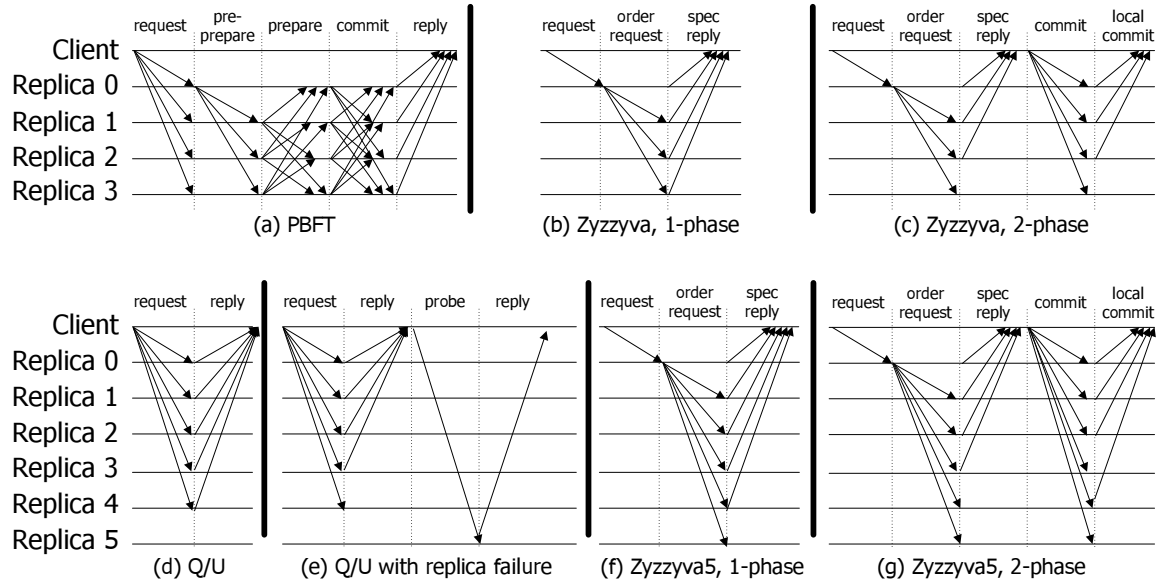


Figure 1: A high-level view of PBFT, Q/U, and Zyzzyva.

hybrid agreement/quorum protocol that shares some of PBFT’s characteristics (a distinguished primary replica and an explicit “view change” to recover from a faulty primary); however, whereas PBFT performs extra work during agreement to ensure it can deal with primary failures, Zyzzyva offloads that extra work to the actual recovery from a primary failure, resulting in a leaner protocol when the fault-free case is the common one. Furthermore, Zyzzyva includes a fast path for unanimous agreement on a request’s sequence number.

In more detail, PBFT requires $3f + 1$ replicas to tolerate f faulty replicas. A client broadcasts its request to all replicas. The primary among them assigns the request a sequence number, and broadcasts that assignment to the replica population in a PREPREPARE message. A backup that receives such an assignment acknowledges it and synchronizes with all other replicas on this assignment by broadcasting a PREPARE message to the population. As soon as a replica has received a quorum of $2f + 1$ PREPARE messages, it promises to commit the request at that sequence number by broadcasting a COMMIT message. When a replica has seen a quorum of $2f + 1$ such promises for the same request and sequence number, it finally accepts that assignment and executes the request in its local state after it has executed all other requests with lower sequence number, sending a REPLY message to the client with the result. A client accepts the result if $f + 1$ replicas send matching REPLY messages, and otherwise retransmits the request. See Figure 1(a) for an illustration.

In contrast, Query/Update (Q/U) is a single-phase quorum-based protocol that tolerates up to f faulty repli-

cas in a population of $5f + 1$. Clients cache replica histories (the request sequence known by a replica), which they include in requests to replicas, and which they update using replies from replicas. These histories allow a replica that receives a client request to optimistically execute it immediately, as long as its request history is reflected in the client’s view. When a client receives replies, as long as a quorum of $4f + 1$ have optimistically executed its request, it completes. Normally a client only contacts its “preferred quorum” of replicas instead of the whole population; if some of the quorum replicas are slow to respond, a client might engage more replicas via a “probe” hoping to complete the quorum. If between $2f + 1$ and $4f$ replies accept the request but others refuse due to a stale replica history, the client infers there exists a concurrent request from another client. Q/U provides a conflict resolution mechanism in which clients back off and later resubmit requests, after repairing the replicas to make them consistent. Figure 1(d) shows the best case for a client’s request, whereas Figure 1(e) illustrates the probing mechanism.

Zyzzyva uses a primary to order requests, like PBFT, and also requires $3f + 1$ replicas to tolerate f faults. Clients in Zyzzyva send the request only to the primary. Once the primary has ordered a request, it submits it in an ORDERREQ message to the replicas, which respond to the client immediately as in Q/U. In failure-free and synchronous executions in which all $3f + 1$ replicas return the same response to the client, Zyzzyva is efficient since requests complete in 3 message delays and, unlike Q/U, write contention by multiple clients is mitigated by the primary’s ordering of requests (Figure 1(b)). When some

replicas are slower or faulty and the client receives between $2f + 1$ and $3f$ matching responses, it must marshal those responses and resend them to the replicas, to convince them that a quorum of $2f + 1$ has chosen the same ordering for the request. If it receives $2f + 1$ matching responses to this second phase, it completes the request in 5 message delays (Figure 1(c)).

To trade off fewer message delays for more replicas in high-jitter conditions, or when some replicas are faulty or slow, the authors propose Zyzyva5, which requires $5f + 1$ replicas and only $4f + 1$ optimistic executions from replicas to progress in 3 message delays—up to f replicas can be slow or faulty and the single-phase protocol will still complete (Figure 1(f)). With fewer than $4f + 1$ replies, Zyzyva5 also reverts to two-phase operation (Figure 1(g)). Finally, Zyzyva’s view change protocol is more heavy-weight and complex than in PBFT, and the authors present results where one replica is faulty (mute) and observe that Zyzyva is slower than PBFT in this situation; however a recent optimization [15] improves Zyzyva’s performance under faults. In this optimization, clients explicitly permit replicas to send a response only after having committed it (as opposed to tentatively). For such client requests, replicas agree on orderings similarly to the second phase of PBFT, and clients need not initiate a second protocol phase.

Each of these protocols achieves high performance by focussing on a specific expected common scenario (failures, latency skew, contention level, etc.). As a consequence, each is ingeniously optimized in a different way. This makes it hard for a developer with a requirement for BFT to choose a high-performance protocol for her specific application. Worse, evaluations of individual protocols in the literature tend to use different scenario parameters. Our aim in this paper is to enable, and perform, “apples to apples” comparison of such protocols as a first step in establishing their performance envelopes.

3 Methodology

We now describe in detail our approach to comparing BFT protocols experimentally. We have built BFTSim, which combines a declarative front end to specify BFT protocols with a back-end simulator based on the widely used ns-2 simulator [1]. This allows us to rapidly implement protocols based on pseudocode descriptions, evaluate their performance under a variety of conditions, and isolate the effects of implementation artifacts on the core performance of each protocol.

Using simulation in this manner raises legitimate concerns about fidelity: on what basis can we claim that results from BFTSim are indicative of real-world performance? We describe our validation of BFTSim in section 3.3 below, where we reproduce published results

from real implementations.

A further concern is the effort of re-implementing a published protocol inside BFTSim, including characterizing the costs of CPU-bound operations such as cryptographic primitives. We report on our experience doing this in section 3.4.

However, a first question is: why use simulation at all? In other words, why not simply run existing implementations of protocols in a real networking environment, or one emulated by a system like ModelNet [23]?

3.1 Why Simulation?

Subject to fidelity concerns which we address in section 3.3 below, there are compelling advantages to simulation for comparing protocols and exploring their performance envelopes: the parameter space for network conditions can be systematically and relatively easily explored under controlled conditions.

There are highly pragmatic reasons to adopt simulation. Many implementations of BFT protocols from research groups are not available at publication time, due to inevitable time pressure. Comparing the performance of protocols based on their published descriptions without requiring re-implementation in C, C++, or Java is a useful capability.

Even implementations that are available vary widely in choice of programming language, runtime, OS, cryptographic library, messaging, thread model, etc., making it hard to identify precisely the factors responsible for performance or, in some cases, to even run under emulation environments such as Emulab due to incompatibilities. For example, Q/U uses SunRPC over TCP as the communication framework while PBFT uses a custom reliability layer over UDP; Q/U is written in C++ while PBFT is written in C; Q/U uses HMAC for authenticators while Zyzyva and PBFT use UMAC. Our results below show that performance is generally either network-bound or dominated by the CPU costs of crypto operations. We can build faithful models of the performance of such implementations based on the costs of a small number of operations, and hence directly compare algorithms in a common framework.

Furthermore, simulation makes it straightforward to vary parameters that are non-network related, and so cannot be captured with real hardware in a framework such as ModelNet. For example, since CPU time spent in cryptographic operations is at present often the dominating factor in the performance of these protocols, we can explore the future effect of faster (or parallel) cryptographic operations without requiring access to faster hardware.

Compared to a formal analytical evaluation, simulation can go where closed-form equations are difficult to derive. Existing literature presents analyses such as fault-

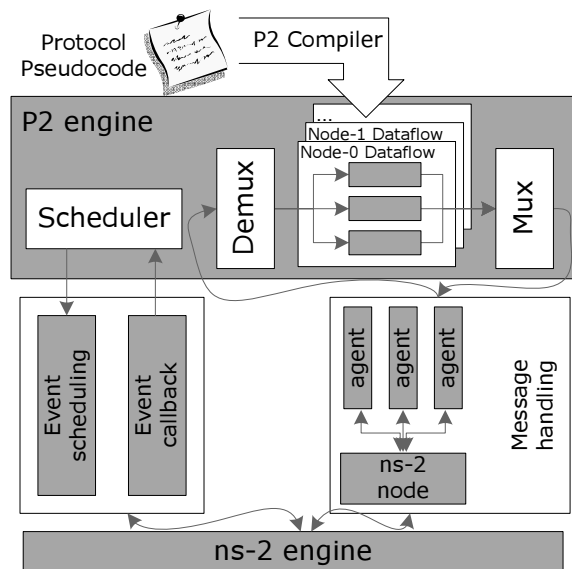


Figure 2: The BFTSim software architecture.

scalability by counting messages exchanged and cryptographic operations performed [9, 14], but it is hard to analyze the dynamic behavior of these protocols, especially for costs intended to be amortized over different sequences of requests. For example, the interaction of client retransmissions, pipeline depth in protocols with a bounded number of requests in flight, and request batching can have a complex effect on response times. Systematic evaluation using faithful simulation can answer these questions with greater ease.

A final motivation for simulation is pedagogical. It is widely believed that BFT protocols are complex to design, implement, and evaluate [8], and that it is hard to understand what aspect of a given protocol gives rise to its performance characteristics. A simple and succinct re-implementation in a declarative language which shows (under simulation) the same performance characteristics as a published C++ implementation is a powerful tool for understanding such issues. We believe that BFTSim may lead the many BFT protocols available to greater accessibility.

3.2 The Design of BFTSim

BFTSim consists of several components (Figure 2). First, protocols are implemented in, and executed by, the front end of the P2 declarative networking engine [18]. P2 allows concise specification and rapid prototyping of BFT protocols, in a manner closely following pseudocode descriptions in publications: we specify the core features of these protocols in a small number of declarative rules.

A rule is of the form “result :- preconditions.” and is interpreted as “the result is produced when all precon-

ditions are met.” For example, consider the following simplified rule for PBFT:

```
initPrePrepare(@A, T, OP, CID) :-
    request(@A, T, OP, CID),
    cachedReply(@A, CID, T1, REPLY), T > T1,
    isPrimary(@A, A, V).
```

It means that when a request tuple arrives at node A from the client with identifier CID, and if the timestamp T is more recent than the last reply sent to the same client (condition $T > T1$), then the primary (condition `isPrimary`) produces a `initPrePrepare` tuple to start the protocol for this request. The location specifier @A is used to specify on which node this particular rule is executed.

P2 compiles such descriptions to a software dataflow graph, which in the case of BFTSim includes elements to capture the timing characteristics of CPU-intensive functions without necessarily performing them. Our hypothesis (subsequently borne out by validation) was that we can accurately simulate existing BFT protocols by incorporating the cost of a small number of key primitives. We started with two: cryptographic operations and network operations (we assume for now that disk access costs are negligible, though modeling disk activity explicitly may be useful in some settings). To feed our simulator with the cost of these primitives, we micro-benchmarked the PBFT and Q/U codebases to find the cost of these primitives with varying payload sizes. Our simulator uses this information to appropriately delay message handling, but we can also vary the parameters to explore the impact of future hardware performance.

In BFTSim, a separate P2 dataflow graph is generated per simulated node, but all dataflow graphs are hosted within a single instance of the P2 engine. P2’s scheduler interacts with the ns-2 engine via ns-2’s event scheduling and callback interfaces to simulate the transmission of messages and to inject events such as node failures. Those interfaces in turn connect with a single ns-2 instance, initialized with the network model under simulation. Message traffic to and from a P2-simulated node is handled by a dedicated UDP agent within ns-2³. Note that both P2 and ns-2 are discrete-event-based systems. We use ns-2’s time base to drive the system. P2’s events, such as callbacks and event registrations, are wrapped in ns-2 events and scheduled via ns-2’s scheduler. BFTSim is currently single-threaded and single-host.

3.3 Validation

In this section we compare the published performance of several BFT protocols with the results generated by our implementation of these protocols in BFTSim under comparable (but simulated) conditions. These results therefore yield no new insight into BFT protocols, rather they

serve to show that BFTSim succeeds in capturing the important performance characteristics of the protocols.

We present a small selection of our validation comparisons for three protocols: PBFT [7], Q/U [3], and Zyzzyva/Zyzzyva5 [14]. PBFT was chosen because it is widely regarded as the “baseline” for practical BFT implementations. Q/U provides variety: it is representative of a class of quorum-based protocols. Finally, the recent Zyzzyva is considered state-of-the-art, and exhibits many high-level optimizations.

For all these protocols, we compare BFTSim’s implementation results to either published results or the protocol authors’ implementations executing in our local cluster. Table 1 lists our validation references. Our validation concentrates on latency-throughput curves for all protocols in typical “ideal” network conditions, as well as under node crashes.

3.3.1 Experimental Setup

We simulated a star network topology, where both client and replica nodes are connected to each other via a hub node. Each link is a duplex link and we set a one-way delay of 0.04ms and bandwidth of 1000Mbps on each link. This gives an RTT of 0.16ms between any pair of nodes, matching our local cluster setup. These values are also similar to those reported by the authors [13] to obtain the Zyzzyva/Zyzzyva5 and PBFT results that we use for validation. Both PBFT and Zyzzyva exploit hardware multicast to optimize the one-to-all communication pattern among replicas. BFTSim accounts for multicast by charging for a single message digest, message send, and authenticator calculation for multicast messages, but currently simulates multiple unicast messages at the ns-2 level.

Since the literature only provides peak throughput results for Q/U, we obtained the authors’ implementation and ran it in our local cluster; each machine has a dual-core 2.8 GHz AMD processor with 4GB of memory, running a 2.6.20 Linux kernel.

3.3.2 Cost of Key Operations

We measured the costs of three primitive operations common to all protocols: calculating the message digest, generating a MAC and authenticator, and sending a message with varying payload sizes. We instrumented the PBFT and Q/U codebases to measure these costs.

To measure the host processing delays required to transmit and receive messages of a given size, we performed a simple ping-pong test between two machines connected by gigabit ethernet. We measured the total time taken to send and receive a response of equal size. We then simulated the same experiment in ns-2 (without BFTSim) to determine the round-trip network delays. For each message size, we then subtracted the simulated net-

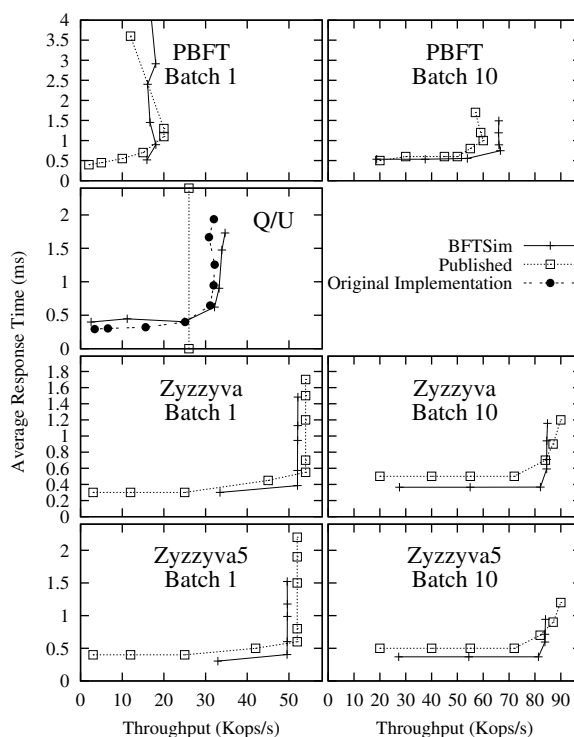


Figure 3: Baseline validation. Note that, to enhance readability, ranges differ for the y axis across protocols (i.e., rows) and for the x axis between batch sizes (i.e., columns).

work delays from the corresponding measured round-trip times to determine the host processing delays. We used linear interpolation to obtain piece-wise linear formulae for estimating the send and receive costs as a function of the message size, with a confidence of 99%.

All protocols use MD5 for calculating message digests⁴. Based on our benchmarking results, we obtained the following linear interpolation to estimate the cost of an MD5 digest over d bytes: $digest(d) = (0.0097d + 0.74)\mu\text{secs}$, with a confidence of 99.9%. We measured the cost of authenticators by varying the number of MAC operations and used this information directly. Note that Q/U uses HMAC while PBFT and Zyzzyva use UMAC for calculating MACs; our validating simulations are parametrized accordingly.

3.3.3 Baseline Validation

Figure 3 presents latency-throughput curves for BFTSim implementations of PBFT, Q/U, Zyzzyva, and Zyzzyva5, compared to our reference sources. We present results with batch sizes of 1 and 10. We executed each protocol under increasing load by varying the number of clients between 1 and 100. Each client maintains one outstanding request at any time. Each point in the graphs represents an experiment with a given number of clients. The

Table 1: Reference sources for our validation and BFTSim implementation coverage for each protocol.

Protocol	Validated Against	Protocol features <i>not</i> present in BFTSim implementations
PBFT	[14]	State transfer, preferred quorums.
Zyzyva/Zyzyva5	[14]	State transfer, preferred quorums, separate agreement & execution.
Q/U	Implementation, [3]	In-line repair, multi-object updates.

knee in each curve marks the point of saturation; typically, the peak throughput is reached just beyond and the lowest response time just below the knee.

First, we note that the trends in the latency-throughput curves obtained with BFTSim closely match the reference for all the protocols we studied. The differences in the absolute values are within at most 10%. Because no latency-throughput curves for Q/U were published, we measured the original implementation in our local cluster⁵. The published peak throughput value is also shown.

The results show that our implementations in BFTSim correctly capture the common case behavior of the protocols, and that BFTSim can accurately capture the impact of the key operations on the peak performance of all protocols we implemented and evaluated.

3.3.4 Validating the Silent Replica Case

In this experiment, we make one of the replicas mute for the duration of the experiment. This experiment exercises important code paths for all protocols. In PBFT, performance may improve in this situation since replicas avoid both receiving and verifying messages from the silent replica. In contrast, Zyzyva’s performance is expected to drop in the presence of a faulty replica since it requires clients to perform the costly second phase of the protocol. In Q/U, performance also drops in the presence of a faulty replica because all requests must be processed by the remaining live quorum of 5 replicas, which increases the load on each of these replicas. In the absence of a faulty replica, each replica handles only $(4f + 1)/(5f + 1)$ -th of the requests, due to Q/U’s preferred quorum optimization.

We present BFTSim’s results along with the published results in Figure 4. Because no published results are available for Q/U with a faulty replica, we measured the original implementation in our cluster in the presence of a silent replica for comparison. We observe that BFTSim is able to closely match the published performance of Zyzyva, Zyzyva5, PBFT, and Q/U in this configuration.

3.3.5 Validating Fault Scalability

In this experiment, we scale the fault tolerance of PBFT, Zyzyva, and Q/U and compare the performance predicted by BFTSim with the published results in the Zyzyva and Q/U papers. At higher values of f , all three protocols need to do more MAC calculations per protocol operation since there are more replicas. PBFT ad-

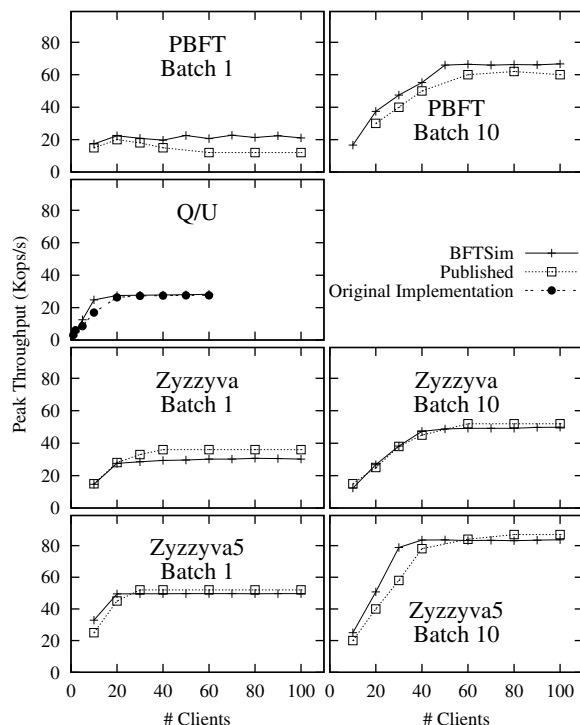


Figure 4: Validation results in the presence of a faulty replica.

ditionally generates more bandwidth overhead at higher values of f since each replica receives $3f$ PREPARE and COMMIT messages. We present results in Figure 5 for values of f between 1 and 3 (we do not validate Zyzyva5 since our reference source offered no measurements for fault scalability). Again, we observe that BFTSim is able to match published results for all protocols for this experiment.

3.4 Implementation Experience

One concern with our approach is the effort required to implement protocols within the framework. Here, we report on our experience implementing the protocols presented.

Our PBFT implementation consists of a total of 148 lines of OverLog (P2’s specification language), of which 14 are responsible for checkpoint and garbage collection, 38 implement view changes, 9 implement the mechanism to fetch requests, and the remaining 87 provide the main part of the protocol. Our implementation of Zyzyva is

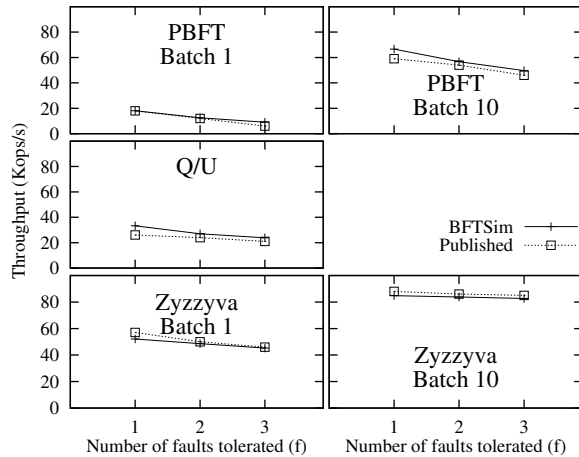


Figure 5: Validation at higher values of f .

slightly more complex: 164 rules, with 13 for checkpoint and garbage collection, 33 for view changes, 24 for handling the second phase, and the remaining 94 for the main part of the protocol. For Q/U, we wrote a total of 88 rules. The key steps in Q/U are represented by the classification (5 rules) and setup (6 rules) phases.

These protocols have served as vehicles for validating the design and fidelity of BFTSim. As a result, their implementation in BFTSim inevitably co-evolved with the simulator itself. However, the basic components (the P2 front end and ns-2 back end) are largely unmodified and thus retain their generality, which of course extends far beyond BFT protocols. We have only modified ns-2 to enable jumbo-frame handling, and we modified and extended P2 in four ways:

- We retargeted P2’s dataflow network stack to use ns-2 agents as opposed to P2’s own congestion-controlled UDP elements.
- We added support for compound (nested) tuples. P2’s original design stayed close to the relational data model, in which tuples are flat. Since support for complex objects in P2’s data model was evolving separately as we were performing this work, we chose to use a simpler but coarser and less efficient approach to structuring data (via explicit nesting of serialized tuples within tuples) for building request batches and the complex view-change messages. Since P2 operates in the virtual time of BFTSim, this inefficiency does not affect our results, only the elapsed time required for our simulations.
- The version of P2 on which we based BFTSim (version 0.8) did not yet support atomic execution of individual rules. Since BFTSim simulates single-threaded, run-to-completion message handlers, we modified the P2 scheduler to complete all processing on outgoing tuples and send them to the ns-2 agent

before any pending incoming tuples were handled.

- We implemented complex imperative computations not involving messaging (such as the view-change logic in PBFT and Zyzzyva) as external plugins (“stages”) in P2.

P2’s language (OverLog) currently lacks higher-order constructs. This makes it cumbersome, for example, to make the choice of cryptographic primitive transparent in protocol specifications, leading to extra OverLog rules to explicitly specify digests and MACs as per the configuration of the particular protocol. As above, this inefficiency does not affect the (simulated) size or timing of messages transmitted, it merely means simulations take longer to complete.

Based on our experience, we feel a more specialized language might reduce protocol implementation effort, but at these code sizes the benefit is marginal. A more immediate win might be to abstract common operations from existing protocols to allow them to be reused, as well as decoupling the authentication and integrity protocol specifications from their actual implementation (i.e., the specific cryptographic tools used to effect authentication, etc.). These are a topic of our on-going research.

4 Experimental Results

BFTSim can be used to conduct a wide variety of experiments. One can compare the performance of different protocols under identical conditions. It is possible to explore the behavior of protocols under a wide range of network conditions (network topology, bandwidth, latency, jitter, packet loss) and under different workloads (distribution of request sizes, overheads, request burstiness). Furthermore, BFTSim allows us to easily answer “what-if” questions such as the impact of a protocol feature or a crypto primitive on the performance of a protocol under a given set of conditions.

Due to time and space constraints, we have taken only a first step in exploring the power of BFTSim, by evaluating BFT protocols under a wider range of conditions than previously considered. Specifically, we evaluate the effects of batching, workload variation (request size), network conditions (link latency, bandwidth, loss, access link variation), and client timer misconfiguration. We also perform an experiment to explore the potential of a possible protocol optimization in Q/U.

When reporting results, we either show throughput as a function of the number of clients used, or report results with sufficiently many clients to ensure that each protocol reaches its peak throughput. When reporting latency, we use a number of clients that ensures that no protocol is saturated, i.e., requests are not queued.

We use a star network topology to connect clients and replicas. Each node, either a replica or a client, is con-

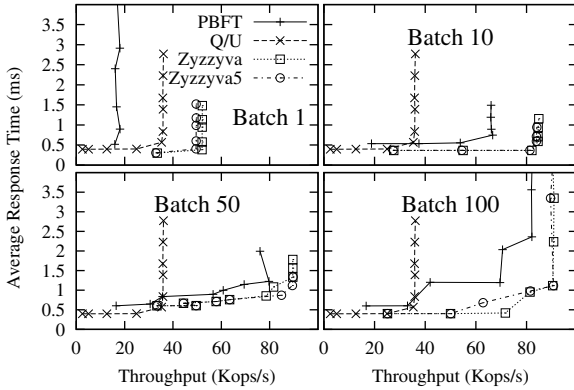


Figure 6: Baseline latency-throughput curves for all protocols, with batch sizes increasing from 1 to 100. Note that Q/U has no batching but appears in all plots for comparison.

nected to a hub of ample capacity using a bidirectional link with configurable latency, bandwidth and loss rate.

4.1 The Effects of Batching

We start with a baseline protocol comparison under typical LAN network conditions (average round-trip time of 0.16 ms, 1 Gbps bandwidth, no packet losses). The requests are no-ops (2-byte payload and no execution overhead). Request batching is used in the agreement-based protocols that support it (all but Q/U). We use the same digest and authentication mechanisms for all protocols (those of the PBFT codebase: MD5 and UMAC).

Figure 6 shows latency-throughput curves for the protocols with increasing batch sizes. As before, each point represents a single experiment with a given number of clients. In agreement-based protocols, the primary delays requests until either a batching timer expires (set to 0.5 ms) or sufficiently many requests (the batch size) have arrived; then, it bundles all new requests into a batch and initiates the protocol. Batching amortizes the messaging overheads and CPU costs of a protocol round over the requests in a batch. In particular, fewer replica-replica messages are sent and fewer digests and authenticators are computed. All three batching-enabled protocols benefit from the technique. Because PBFT has the highest overhead, it enjoys the largest relative improvement from batching: its peak throughput increases by a factor of four.

The Zyzzyva variants are more efficient under the given network conditions, requiring fewer messages and crypto operations than PBFT. As a result, though still fastest in absolute terms, they derive a smaller relative benefit from batching. Furthermore, as batch size increases, the differences between protocols shrink significantly; the Zyzzyva variants become indistinguishable

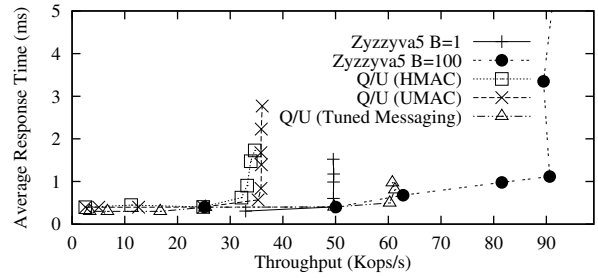


Figure 7: Hypothetical optimized Q/U. Our hypothetical Q/U competes with Zyzzyva5 without batching, but is still inferior to batched Zyzzyva5. We also show Q/U using HMAC, as in its original implementation validated in Figure 3.

with only moderate batching, whereas PBFT achieves throughput within 10% of Zyzzyva at a batch size of 100.

4.1.1 Why is Q/U Worse?

Figure 6 shows that Q/U has significantly lower peak throughput than Zyzzyva5, even with no batching and despite the absence of write contention in our workload. This is somewhat surprising, because in the absence of contention, Q/U only requires a single phase of direct client-replica communication to complete, whereas Zyzzyva5 must relay all requests from clients to replicas via the primary.

We hypothesized that one reason for Q/U’s lower throughput is the size of the messages it requires. Q/U replicas include their recent object history in each response⁶, and clients send with their requests a vector of such replica histories. This is an important safety requirement in Q/U; it enables non-faulty replicas to execute an operation in a consistent order without any replica-to-replica coordination. In contrast, Zyzzyva replicas include only digests of their history with response messages, and clients send no history with their requests, because the primary orders requests.

We performed an experiment to see how a hypothetical version of Q/U would perform that is optimized for message size. Our hypothetical Q/U variant is assumed to send Zyzzyva-style history digests as a vector in Q/U requests, and sends a single history digest in replica responses. The intuition is that, without faults or write contention, history digests are sufficient to convince a replica to execute an operation at a given point in the history, and replicas need exchange full histories only when they detect an inconsistency. We charge Q/U with Zyzzyva’s ORDERREQ-sized messages with respect to digest computation and message transmission, but leave MAC costs unchanged. This experiment is an example of how BFTsim can be used to quickly ask “what-if” questions to explore the potential of possible protocol opti-

mizations.

Figure 7 shows the result. As expected, our hypothetical Q/U is competitive with Zyzzyva5 at a batch size of 1. However, our variant of Q/U is still no match for Zyzzyva5 at a large batch size in terms of peak throughput (though it has slightly lower latency). This seems counter-intuitive at first glance, since Q/U has no “extraneous” traffic to amortize, only client requests and replica responses. However, Q/U’s lack of a primary that orders and relays requests to backups incurs extra computation on the critical path: a Q/U replica computes $4f$ MACs to send a response to a client and verifies $4f$ MACs when receiving a request from it, whereas the Zyzzyva5 primary computes and verifies one MAC for each request in a batch of b requests, plus a single authenticator for each ORDERREQ message containing $4f$ MACs for a total of $2+8f/b$ MACs per request in a batch. As b increases, Zyzzyva5 tends toward slightly more than 1 MAC generation and 1 MAC verification per client request compared to $8f$ MAC operations in Q/U.

4.1.2 Summary

Batching helps agreement-based protocols to achieve better performance by amortizing their protocol costs over a batch of requests. As we increase the batch size, the importance of protocol efficiency diminishes; as a result, the throughput of PBFT approaches that of Zyzzyva. The results of our experiments also predict that Q/U could benefit significantly from optimizations to reduce its message sizes.

4.2 Varying the Workload

We now turn to study the performance of BFT protocols under varying request sizes. The network conditions remain as above. Figure 8 shows latency-throughput graphs for all protocols when the size of the request (and response) ranges from 2 bytes (as in the experiments above) up to 8 kbytes. Request sizes in this range do occur in practice. Whereas many SQL workloads are reported to have request sizes of around 128 bytes, applications like block storage use larger requests (for example, Hendricks et al. [10] discuss requests containing erasure-coded disk block fragments with sizes of about 6 Kb). We obtained these graphs by choosing a number of clients such that the latency-throughput curve was beyond its characteristic “knee,” i.e., the system was saturated (in this set of experiments, between 1 and 80 clients were required). It is striking how increasing payloads diminish the differences among the protocols. Whereas at 2-byte payloads, non-batched PBFT has a little more than a third the throughput of Zyzzyva and Q/U about half, at 8 kbyte payloads all protocols are very close. With batching, PBFT starts out closer to Zyzzyva but, again, the difference vanishes at higher

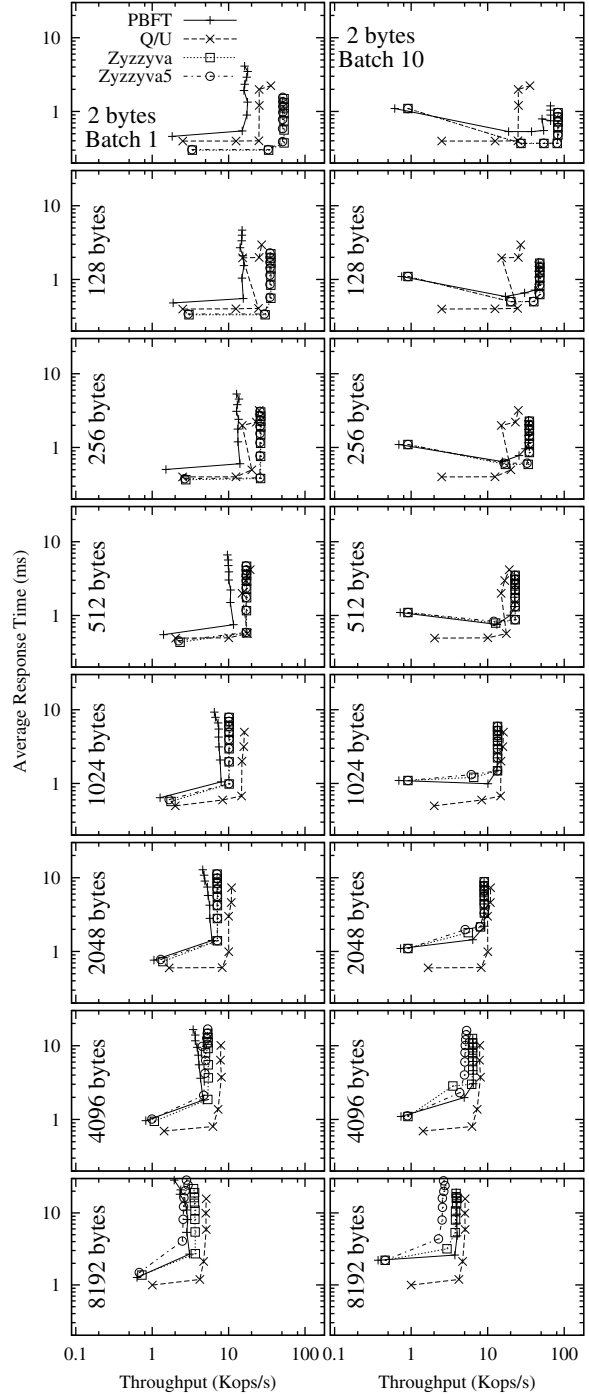


Figure 8: Latency-throughput plots for increasing request payload sizes, without batching (left) and with batch size 10 (right). Note that both axes are in logarithmic scale, to show better the relative performance differences of the protocols at different scales.

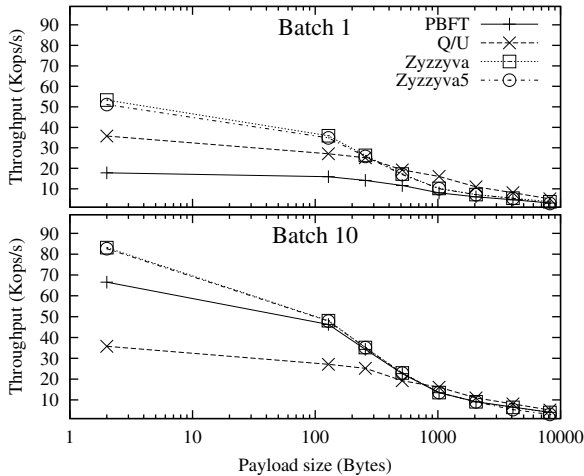


Figure 9: Peak throughput vs. request size. Top: no batching, bottom: batch size of 10. Q/U has no batching but appears in both plots for comparison. Note that the x axis is logarithmic.

payload sizes as throughputs degrade dramatically. The reason is that with increasing request size, the per-byte processing costs and network transmission delays start to dominate per-request costs, which increasingly masks the differences among the protocols.

Though helpful in identifying the latency-throughput trade-offs, the plots in Figure 8 make it difficult to see the trends in peak throughput or latency below saturation. We show in Figure 9 the peak throughput as a function of the request size, and in Figure 10 the mean response time below protocol saturation. Peak throughput trends are clearer here; they are consistent with the large increase in transmission and digest computation costs resulting from larger request sizes.

Interestingly, Q/U appears more robust to increasing payload sizes than the other protocols, and exhibits the least steep decline in throughput. At high payload sizes, Q/U matches the throughput of Zyzzyva and Zyzzyva5, even when these protocols use batching. The reason is that Q/U's messages, which have a larger base size due to the history they contain, are increasingly dominated by the payload as the request size increases. Moreover, at large request sizes, the throughput is increasingly limited by per-byte processing costs like MAC computations and network bandwidth, making batching irrelevant.

Figure 10 shows the average response time for increasing request sizes below system saturation. PBFT with batching and Q/U provide the lowest response times. As payload sizes increase, response times with Zyzzyva and Zyzzyva5 suffer, because Zyzzyva and Zyzzyva5 send a full copy of all requests in a batched ORDERREQ message from the primary to the backups. In contrast, PBFT

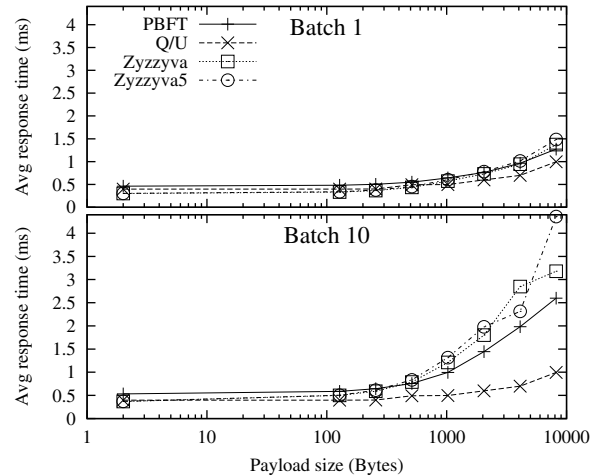


Figure 10: Minimum of per-experiment average request response time vs. request size. Top: no batching, bottom: batch size of 10. The system is *not* saturated (i.e., these are not the corresponding response times for Figure 9). Q/U has no batching but appears in both plots for comparison. Note that the x axis is logarithmic.

only sends message digests in its batched PREPREPARE message, leaving it to the client to transmit the request itself to the backups. The cost of transmitting and authenticating the request content is typically spread over all clients in PBFT, instead of concentrated at the primary in Zyzzyva⁷.

There is no inherent reason why Zyzzyva might not benefit from an optimization similar to that of PBFT under the given network conditions; however, this optimization would increase the bandwidth consumption of the protocol (since all clients will have to send requests to all replicas, as with PBFT) and possibly reduce its ability to deal gracefully with clients who only partially transmit requests to replicas. The best approach may depend on the type of network anticipated. For instance, in networks where multicast is available (e.g., enterprise settings where local multicast deployment tends to be more common), the network component of the overhead caused by large payloads—but not the computation component—may be low.

4.3 Heterogeneous Network Conditions

Next, we place one replica behind a slow link, in order to introduce an imbalance among the replicas. PBFT, Q/U and Zyzzyva5 do not require all replicas to be in sync to deliver peak performance, therefore we expect their performance to be unaffected by heterogeneity. In contrast, we expected Zyzzyva's performance to degrade, because the protocol requires a timely response from all replicas to be able to complete a request in a single phase.

Figure 11 shows results for throughput and aver-

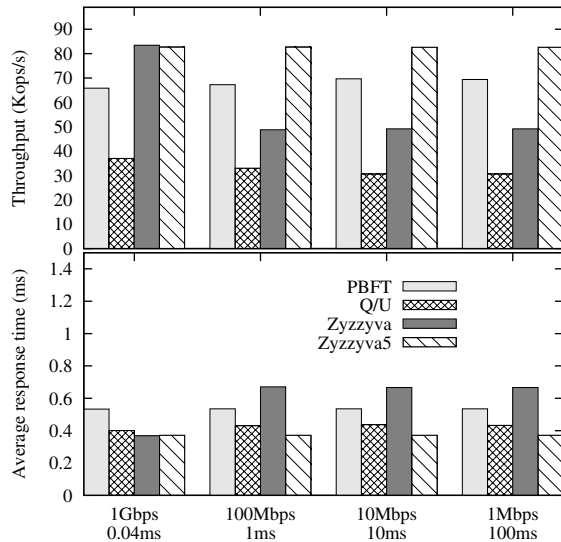


Figure 11: Impact of a replica behind a low bandwidth and high delay link. We vary the characteristics of the slow link, as shown on the x -axis. We set batchsize to 10.

age response time. All but the constrained link in this experiment have 1Gbps bandwidth and 0.04ms latency, whereas the constrained link has the characteristics shown on the x axis. The leftmost configuration is homogenous and included for comparison.

The throughput and response time of Zyzzyva is strongly affected by the presence of a replica with a constrained network link. Zyzzyva must adapt its timers⁸ and eventually switch to two-phase operation as the response time of a slow replica worsens. The throughput of Zyzzyva drops as soon as the imbalance is introduced, because clients start the second phase of the protocol to complete their requests, instead of waiting for the slowest replica to complete the first phase. Consequently, the average response time for Zyzzyva also increases when there is an imbalance. While similar to the mute replica results in Section 3.3.4, our results show that Zyzzyva's throughput is sensitive even to small differences in network latency between a client and the replicas.

Q/U's throughput decreases slightly once there is a slower replica. The reason is that when clients initially include the slow replica in their preferred quorum, they may time out and subsequently choose a quorum that excludes the slow replica. As a result, the remaining replicas experience more load and throughput decreases.

The throughput of PBFT improves slightly with increased heterogeneity, because replicas receive fewer messages from the slow replica, which saves them message receive and MAC verification costs. Note that during sustained operation at peak throughput, the messages from the slow replica are queued at the routers adjacent

to the slow link and eventually dropped. Of course, this loss of one replica's messages does not affect the protocol's operation and it actually increases protocol efficiency slightly.

4.4 Wide-area Network Conditions

Next, we explore the performance of BFT protocols under wide-area network (WAN) conditions, such as increased delay, lower bandwidth and packet loss. Such conditions are likely to arise in deployments where clients connect to the replicas remotely or when the replicas are geographically distributed.

Before presenting the results, we briefly review how the different protocols deal with packet loss.

4.4.1 Background

PBFT In PBFT, if no response to a request arrives, the client eventually times out and retransmits the request to the replicas. A backup replica, upon receiving such a retransmission, forwards it to the primary, assuming it must have missed it. If the primary has already sent a PREPREPARE for that retransmitted message, it does not do anything. However, this could trigger view changes if the original message was lost. To address this problem, PBFT replicas periodically (every 150 ms) multicast small status messages that summarize its state. When another replica notices that the reporting replica is missing messages that were sent in the past, it re-sends these messages [5].

Zyzzyva/Zyzzyva5 In Zyzzyva, if a backup replica receives an ORDERREQ message for seq i from the primary while it expects $k < i$ (suggesting a hole in the history), it sends a FILLHOLE message to the primary. The primary retransmits the missing messages. Loss of one such ORDERREQ message may cause a backup replica to send a FILLHOLE message to the primary for every future ORDERREQ until the hole is patched. This may cause the primary to experience additional load [14].

Q/U In Q/U, replicas inspect the histories contained in request messages to see if they have missed a prior request. If so, they request information from other replicas about the latest state of the corresponding objects. Otherwise, the protocol relies on request retransmission by clients to recover from packet losses.

4.4.2 Results

For the experiment, we configured replicas and clients with different link delay, bandwidth, and uniform random packet loss. Recall that clients and replicas are connected to a common hub via bi-directional links in a star topology. We simulate three wide area configurations as described in Table 2. 50 clients send requests containing 2-byte no-op requests.

Table 2: Wide-area experiment configurations, as shown in Figure 12.

Configuration	Client links (c)	Replica links (r)
Left	30ms, 10Mbps	1ms, 1Gbps
Middle	30ms, 10Mbps	5ms, 100Mbps
Right	30ms, 10Mbps	10ms, 100Mbps

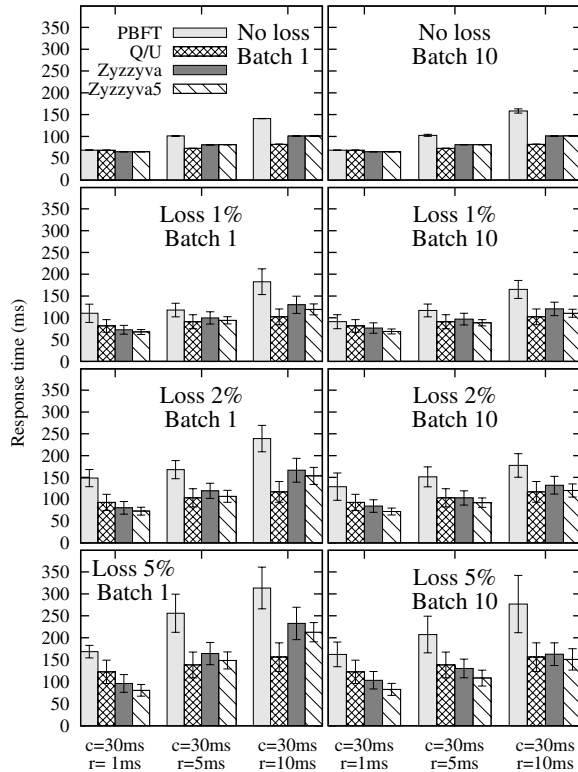


Figure 12: Three configurations on the x axis, average response latency in ms on the y axis. The error bars indicate one standard deviation around the average.

The client retransmission timeout was set based on the expected response time of each protocol. Because the request messages are very small and incur no execution overhead, we approximated the expected response time by adding the link delays on the protocol execution path. Let c be the client link latency and r the replica link latency (both one-way). In the absence of losses and faulty replicas, we expect PBFT to complete a request in time $(2c + 8r)$, Q/U in $(2c + 2r)$, and Zyzzyva/Zyzzyva5 in $(2c + 4r)$. Q/U offers the lowest expected delay followed by Zyzzyva and Zyzzyva5 and then PBFT. We set the client retransmission timeout to this estimate for each protocol, plus 10ms. Figure 12 shows the average response time for all four protocols with and without batching, varying link latencies and loss rate; packet loss is uniform random and affects all links equally.

We make three observations. First, even though Q/U

completes requests in a single phase with the contention-free workload used in this experiment, its response times are not better than those of the hybrid protocols when the replica-to-replica latencies are low (i.e., $r=1$ ms). Once again, the reason is the larger request messages required by Q/U. The extra latencies for transmitting and computing digests for the larger Q/U messages compensate for the extra latencies required for inter-replica communication in the agreement protocols. However, with increasing replica-to-replica latencies, the inter-replica communication significantly adds to the latency of the agreement protocols. Q/U's latencies, on the other hand, increase to a lesser extent because it relies on inter-replica communication only to fetch the current object state after a replica misses a request.

Second, Zyzzyva5 has slightly lower response times than Zyzzyva under message loss. The reason is that Zyzzyva5 requires only $4f + 1$ responses from its $5f + 1$ replicas, while Zyzzyva requires responses from all of its $3f + 1$ replicas to complete a request in a single phase.

Third, batching tends to improve the average latency under losses. Because batching reduces the number of messages per request, it reduces the probability that a given request is affected by a loss, thus reducing the average latency.

4.5 Clients with Misconfigured Timers

The goal of the next experiment is to understand how sensitive existing protocols are to faulty clients. Under contention, it is well known that the performance of quorum protocols like Q/U suffers. In fact, Q/U may lose liveness in the presence of faulty clients that do not back off sufficiently during the conflict resolution phase. The performance of agreement-based protocols like PBFT and Zyzzyva, is believed to be robust to faulty client behavior. The goal of our experiment is to test the validity of this hypothesis. Note that all BFT protocols preserve the safety property regardless of the behavior of clients.

We misconfigured some of the clients' retransmission timers to expire prematurely. The experiments used a LAN configuration, with round-trip delay of 0.16ms on all links (0.04ms one-way for each node-to-hub link). With 100 clients, the average response times are 3ms for Q/U, 1.5ms for PBFT with a batch size of 10, and 1.15ms for Zyzzyva and Zyzzyva5 with a batch size of 10.

We chose four settings of misconfigured timers—0.5ms, 1ms, 1.5ms, and 2ms. This choice of timers allows us to observe how the protocols behave under aggressive retransmissions by a fraction of the clients (between 0 to 25 out of a total of 100 clients.) A client with a misconfigured timer retransmits the request every time the timer expires without backing off, until a matching response arrives. Figure 13 shows the throughput of the protocols as a function of the number and settings of mis-

configured clients, with and without batching. Misconfigured timers affect the throughput of the protocols due to the extra computations of digests, MACs and transmission costs incurred whenever a protocol message must be retransmitted⁹.

Our results show that all protocols are sensitive to premature retransmissions of request messages. This is expected, because premature retransmissions add to the total overhead per completed request.

PBFT and Zyzyva replicas assume a packet loss when they receive a retransmission: PBFT backups forward the message to the primary when they receive a request for the second time, while Zyzyva/Zyzyva5 backups forward the request to the primary immediately because a client is expected to send a request to backups only upon a timeout. A Zyzyva primary, upon receiving such a forwarded request message from a backup replica, responds with the ORDERREQ message. A PBFT primary responds with a PREPREPARE to the backup only if it has never seen the request, otherwise it ignores the retransmission. By retransmitting a request to all backups, a misconfigured client causes the primary of both PBFT and Zyzyva/Zyzyva5 to receive additional messages from the backups.

A Q/U replica, upon receiving a retransmission, responds with the cached response if the request has executed, or else processes the request again. No additional replica-to-replica communication is necessary under a contention-free workload.

The relative impact of clients with misconfigured timers is more pronounced with batching. With batching, each individual retransmitted request can cause the retransmission of protocol messages for the entire batch that contained the original request. As a result, the aggregation of protocol messages that occurs in normal protocol operation does not occur for retransmitted requests.

5 Related Work

While there has been considerable work on simulators for networks and P2P systems, we are aware of relatively little work that attempts to model both CPU performance and network characteristics, though we note that a similar technique was used by Castro [5] to build an analytical model of the PBFT protocol.

Systems like WiDS [17], Macedon [21], and its successor MACE [12] allow distributed systems to be written in a state-machine language, which can then be used to generate both a native code implementation and drive a simulator, which also executes “real” code.

BFTSim takes a different approach, simulating both the message exchange and the CPU-intensive operations. This allows easy exploration of the effect of CPU performance for crypto operations, and P2’s declarative spec-

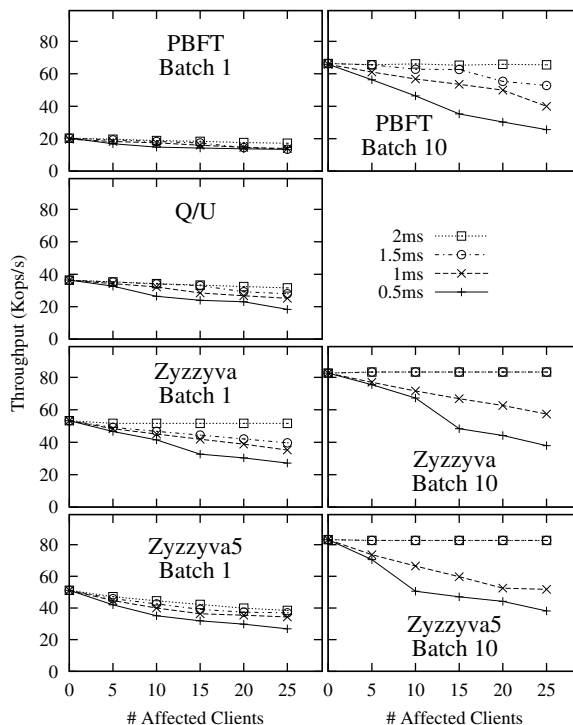


Figure 13: Clients with misconfigured retransmission timers. We vary the number of such clients on the x -axis (100 clients total).

ifications are an order of magnitude more concise than MACE (which itself is considerably more concise than a manual implementation in Java or C++). The disadvantage with BFTSim is that BFT protocols thus specified cannot be executed “for real” in a production system without extensions to P2, and would likely result in a less efficient implementation due to P2 currently generating software dataflow rather than native code.

We chose our BFT protocols (PBFT, Q/U, and Zyzyva) to provide good coverage for BFTSim in order to evaluate its effectiveness as well as provide interesting comparisons. However, recent research has produced a slew of new protocols, which we intend to examine.

To take one example, Cowling et al.’s HQ protocol [9] ingeniously combines quorum and consensus approaches. HQ is a two-round quorum protocol in the absence of write conflicts and requires $3f + 1$ replicas. Replicas optimistically choose an ordering of requests (a *grant*) and notify the client, which collects a quorum of $2f + 1$ such grants and in a second round returns the collected grants (a *writeback*). Replicas detect contention by observing the set of grants in the writeback, and resort to Byzantine consensus for resolution rather than exponentially backing off as in a pure quorum system. As a result, HQ improves upon PBFT in low-concurrency settings, while resolving concurrency conflicts at a lower

expected latency than Q/U. Published results for HQ so far are only for fault-free settings with low-latency, high-bandwidth links.

Part of our agenda in this paper is to argue for the comparison of distributed algorithms on a level playing field in a variety of realistic, but different, scenarios. Only then can developers select appropriate algorithms for the particular tradeoffs at hand. Such a shift in thinking has been recently recognized in processor architecture, where it is termed “scenario-oriented design” [20].

6 Future Work and Conclusions

In this paper we recognize that, though bold, new moves have been made towards designing and implementing efficient, safe, and live Byzantine-fault tolerant replicated state machines, little has been done to look under the covers of those protocols and to evaluate them under realistic imperfect operating conditions. We argue that a simulation framework in which fundamentally different protocols can be distilled, implemented, and subjected to scrutiny over a variety of workloads, network conditions, and transient benign faults, can lead to the deeper understanding of those protocols, and to their broad deployment in mission-critical applications.

Our first contribution has been BFTSim, a simulation environment that couples a declarative networking platform for expressing complex protocols at a high level (based on P2), and a detailed network simulator (based on ns-2) that has been shown to capture most of the intricacies of complex network conditions. Using BFTSim, we have validated and reproduced published results about existing protocols, as well as the behavior of their actual implementations on real environments. We feel confident that this will encourage the systems community to look closely at published protocols and understand (and reproduce!) their inherent performance characteristics, with or without the authors’ implementation, without unreasonable effort.

Second, we have taken some first steps towards this goal with three protocols, PBFT, Q/U, and Zyzzyva/Zyzzyva5. We have identified some interesting patterns in how these protocols operate:

- One-size-fits-all protocols may be tough if not impossible to build; different performance trade-offs lead to different design choices within given network conditions. For instance, PBFT offers more predictable performance and scales better with payload size compared to Zyzzyva; in contrast, Zyzzyva offers greater absolute throughput and is significantly more robust in wider-area, lossy networks.
- In the contention-free workloads we study, Q/U can demonstrate its strengths in particular as payload sizes grow and replica-replica latencies increase,

compared to all competing protocols. This opens up an intriguing question: what if Q/U were less vulnerable to write contention? An overly simple assessment might argue that Zyzzyva is roughly equivalent to Q/U with an explicit preserializer of requests, which ensures that no write contention occurs in the absence of Byzantine faults [22]. It may be productive to assess to what extent this similarity is only superficial and, if not, what benefits one might gain from building a protocol from scratch, versus engineering a safe composition of existing protocols. Alternatively, a new protocol that shares Q/U’s optimistic one-phase execution with HQ’s efficient contention resolution may become appealing, especially for large-request workloads.

- Timeouts should be set very carefully. This should come as no surprise. However, some protocols are more vulnerable to timeout misconfigurations than others. With Zyzzyva, the ability to complete a request with a single phase offers spectacular opportunities for high throughput, but misconfiguration of the timeout, or the inevitable jitter in wide-area deployments, can rob the protocol of its benefits. In contrast, Q/U tolerates wider timer misconfigurations, but has less to lose in absolute terms.

Although we are confident that our approach is promising, the results described in this paper only scratch the surface of the problem. We have not yet assessed the particular benefits of reliable transports, especially in the presence of link losses; we have only studied a simplified notion of multicast and have yet to study data-link broadcast mechanisms used by some protocols; we have only explored relatively simplistic, geographically constrained topologies instead of more complex, wide-area topologies involving faraway transcontinental or transoceanic links; and we have limited ourselves to the relatively closed world of Byzantine-fault tolerant replicated state machines. Removing these limitations from BFTSim is the subject of our on-going research.

Moving forward, we are expanding the scope of our study under broader workload conditions (varying request execution costs, cryptographic costs, heterogeneous computational capacities), network conditions (skewed link distributions, jitter, asymmetric connectivity), and faults (flaky clients, denial of service attacks, timing attacks). We particularly hope to extract the salient features of different protocols (such as PBFT’s and Zyzzyva’s ways of dealing with client requests, or Zyzzyva’s and Q/U’s similarities modulo request pre-serialization), as well as expand to incorporate storage costs, and bandwidth measurements. Finally, we hope to stimulate further research and educational use by making BFTSim publicly available, along with our implementations of BFT protocols.

Acknowledgments

We are particularly indebted to Michael Abd-El-Malek, Miguel Castro, Allen Clement, and Ramakrishna Kotla for their help in sorting through protocol and implementation details needed by our work. We are grateful to Byung-Gon Chun and Rodrigo Rodrigues for their comments on earlier drafts of this paper. Finally, we heartily thank our shepherd, Miguel Castro, and the anonymous reviewers for their deep and constructive feedback.

References

- [1] The NS-2 Project. http://nsnam.isi.edu/nsnam/index.php/Main_Page, Oct. 2007.
- [2] M. Abd-El-Malek. Personal communication, 2007.
- [3] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.
- [4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.
- [5] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, Jan. 2001. Technical Report MIT/LCS/TR-817.
- [6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 1999.
- [7] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.
- [8] B.-G. Chun, S. Ratnasamy, and E. Kohler. A Complexity Metric for Networked System Designs. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, 2008.
- [9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2006.
- [10] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2007.
- [11] F. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: Caveat Emptor. In *Proceedings of USENIX Hot Topics in System Dependability (HotDep)*, 2007.
- [12] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of ACM Programming Languages Design and Implementation (PLDI)*, 2007.
- [13] R. Kotla. Personal communication, 2007.
- [14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2007.
- [15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *University of Texas at Austin, Technical Report: UTCS-TR-07-40*, 2007.
- [16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- [17] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an Integrated Toolkit for Distributed System Development. In *Proceedings of USENIX Hot Topics in Operating Systems (HotOS)*, 2005.
- [18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.
- [19] B. M. Oki and B. H. Liskov. Viewstamped Replication: a General Primary Copy. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.
- [20] S. M. Pieper, J. M. Paul, and M. J. Schulte. A New Era of Performance Evaluation. *IEEE Computer*, 40(9), 2007.
- [21] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, 2004.
- [22] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. Conflict-free Quorum based BFT Protocols. Technical Report 2007-2, Max Planck Institute for Software Systems, 2007.
- [23] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2002.
- [24] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology—EUROCRYPT 05*, 2005.

Notes

¹We use the terms “latency” and “response time” interchangeably when referring to protocol performance.

²The Paxos protocol [16]—also concurrently discovered as Viewstamped Replication [19]—forms the basis of most fault-tolerant consensus mechanisms, in crash-fault or Byzantine-fault settings.

³Support for jumbo UDP frames was incomplete in the released version of ns-2. We had to enable jumbo-frame handling and IP fragmentation for the large-sized UDP messages of our protocols. However, we have induced no extra delays due to IP fragmentation or reordering. Furthermore, BFTSim does not currently simulate network congestion.

⁴There is a minor error in the Zyzzyva publication, implying that Zyzzyva uses AdHash for message digests and MD5 for MACs. We have since confirmed that, as with the PBFT codebase on which Zyzzyva is built, MD5 is used for digests (with AdHash-MD5 only for incremental state digests) and UMAC for MACs. Note that MD5, either in one-shot mode or incremental AdHash form, is no longer considered collision-resistant [24]; we use it here for validation purposes only.

⁵We were at first unable to reproduce reasonable performance results with the Q/U implementation, a similar experience to other research groups [2]. We fixed a bug with DNS resolution in the Q/U codebase (acknowledged and incorporated in release 1.2 of Q/U) that removed the problem.

⁶History is an ordered set of *candidates*, where each candidate is a pair of logical timestamps. A logical timestamp is represented as $\langle \text{TIME}, \text{BARRIERFLAG}, \text{CLIENTID}, \text{OPERATION}, \text{OHS} \rangle$, where OHS is the object history set.

⁷Note that PBFT offers a runtime parameter for including entire requests within batches; the default configuration of the codebase turns this option off.

⁸Note that we implement an adaptive timer mechanism for clients [13] used in, but not described in, the Zyzzyva publication. Once a client receives $2f + 1$ matching responses it starts an adaptive timer, initialized to a low value, and starts the second phase if this timer expires before receiving the full $3f + 1$ responses. If a client receives $3f + 1$ responses before completing the second phase, it increases the adaptive timer to avoid starting the second phase too early next time. If the second phase completes sooner, the timer is reset to the initial low value.

⁹The original PBFT implementation appears to implement an optimization that caches the digests of transmitted messages; therefore, digests do not have to be recomputed when retransmitting a message. We have not yet implemented this optimization in our version of PBFT.

Uncovering Performance Differences among Backbone ISPs with Netdiff

Ratul Mahajan Ming Zhang Lindsey Poole Vivek Pai
Microsoft Research *Princeton University*

Abstract – We design and implement Netdiff, a system that enables detailed performance comparisons among ISP networks. It helps customers and applications determine, for instance, which ISP offers the best performance for their specific workload. Netdiff is easy to deploy because it requires only a modest number of nodes and does not require active cooperation from ISPs. Realizing such a system, however, is challenging as we must aggressively reduce probing cost and ensure that the results are robust to measurement noise. We describe the techniques that Netdiff uses to address these challenges.

Netdiff has been measuring eighteen backbone ISPs since February 2007. Its techniques allow it to capture an accurate view of an ISP's performance in terms of latency within fifteen minutes. Using Netdiff, we find that the relative performance of ISPs depends on many factors, including the geographic properties of traffic and the popularity of destinations. Thus, the detailed comparison that Netdiff provides is important for identifying ISPs that perform well for a given workload.

1 Introduction

Knowledge of the performance characteristics of ISP networks is highly valuable. It can enable customers and applications to make informed choices regarding which ISP(s) to use for their traffic. These choices are important because the performance of distributed applications depends heavily on the network paths that are used.

Shedding light on ISP performance can also improve overall network infrastructure. Application performance in the Internet depends collectively on multiple ISPs. Unfortunately, the inability to differentiate individual ISPs' performance creates little incentive for ISPs to resolve problems and promote internal innovation [24]. In response, researchers have proposed radical network architectures based on ISP accountability, overlays or customer-directed routing [2, 5, 6, 24, 32, 41]. However, we believe that simply providing visibility into ISPs' performance creates the right incentives. For instance, no particular ISP is motivated to act if studies report that the average latency in the Internet is 60 ms. If instead, studies report that the average latency for the customers of an ISP is twice that for the customers of competitors, market forces will motivate the ISP to improve its performance.

It is thus surprising that the problem of systematically understanding how well various ISPs deliver traffic has

received little attention, especially in the research community. To our knowledge, there has been only one commercial effort [20], whose limitations we discuss in the next section. Today, customers of ISP networks are often in the dark about which ISPs are better and if the higher price of a particular ISP is justified by better performance [25, 26, 35, 42]. A common method for customers to obtain this information is by asking each other about their experiences [25, 26, 42]. Similarly, distributed applications are unaware of how the choice of ISP impacts performance. Even if they use measurements to learn this [3, 14], they cannot predict the performance for ISPs to which they do not directly connect.

Motivated by the observations above, we consider the task of comparing the performance of ISP networks, both in the recent past and over longer time periods. We focus on large ISPs that form the backbone of the Internet. Collectively, these ISPs carry most of the application traffic in the Internet. Their customers include content providers, enterprises, universities, and smaller ISPs.

We first identify the important requirements for a system to compare ISPs. These requirements govern how the measurements should be conducted and analyzed. A key requirement is to quantify performance in a way that is relevant to customers and applications. This implies, for instance, that we measure the performance of paths that extend to destination networks, rather than stopping where the paths exit the ISP's network. The latter is common in service level agreements (SLAs) of ISPs today, but it is less useful because application performance depends on the performance of the entire path. Other requirements include enabling a fair comparison among ISPs, by taking into account the inherent differences in their sizes and geographic spreads, as well as helping ISPs improve their networks.

We then design and implement a system, called Netdiff. It is composed of a modest number of measurement nodes placed inside edge networks and does not require active cooperation from the ISPs themselves. It is thus easy to deploy. There are several challenges in making such a system practical, however. For instance, we must aggressively control probing overhead, which can be prohibitive if implemented naively; we devise a novel set covering-packing based method to systematically eliminate redundant probes. The body of the paper discusses these challenges in detail and describes how we address

them. Our current implementation measures ISP performance in terms of path latency, which is a basic measure of interest for most applications. We are currently extending Netdiff to other relevant measures.

Netdiff has been operational on PlanetLab since February 2007. It currently measures eighteen backbone ISPs. We find that its methods are highly effective. For instance, it reduces the probing overhead by a factor of 400 compared to a naive probing technique. We also find that application performance is closely correlated with its inferences. An informal case study involving a real customer confirms that Netdiff is useful for ISPs' customers.

To further demonstrate the usefulness of Netdiff, we use it to compare the performance of the ISPs that it measures. We find that traffic performance can vary significantly with the choice of ISP, but no single ISP is best suited for all types of workloads. For instance, some ISPs are better at delivering traffic internationally, while others are better for domestic traffic; and some ISPs are better for traffic originating in certain cities, while others are better for certain other cities. We also find that the performance of paths internal to an ISP, which form the basis of typical SLAs, do not reflect end-to-end performance. Thus, selecting an ISP is a complex decision, and the detailed comparison enabled by Netdiff can be very helpful.

This paper considers only the technical aspects of ISP performance comparison and ignores other aspects such as acceptability to ISPs. Ultimately, the success of our effort depends on non-technical aspects as well and we have started investigating them. Encouragingly, in other domains, vendors have accepted performance comparison and even actively participate in the process [36, 39].

2 Related Work

In this section, we place our work in the context of existing methods to compare ISPs and measure networks.

2.1 Methods to Compare ISPs Today

There are two main options available today to customers who want to compare ISPs' performance.

Service-level agreements (SLAs) Many ISPs offer an SLA that specifies the performance that customers can expect. These SLAs are typically not end-to-end and specify performance only within the ISP's network. For instance, an SLA may promise that 95% of traffic will not experience latency of more than 100 ms inside the ISP's network. A few providers also offer "off-net" SLAs in which performance is specified across two networks – the ISP's own network and that of some of its neighbors.

Today's SLAs have two shortcomings when using them to compare ISPs. First, application performance depends on the entire path to the destination and not just on a particular subpath. As such, ISPs with better SLAs may not

necessarily offer better performance (Section 7.2). Second, because they are independently offered by each ISP, SLAs make comparisons among ISPs difficult. Some SLAs may mention latency, some may mention loss rates, some may mention available capacity, and yet others may mention a combination of metrics. Even with comparable measures, further difficulties in comparison arise due to differences in the size and geographic spread of each ISP. For instance, is a 100-ms performance bound for an ISP with an international network better or worse than a 50-ms bound for an ISP with a nation-wide network? Our work uses measures that can be used to compare ISPs regardless of differences in their networks.

Third-party systems There exist systems that allow multihomed customers to select the best ISP for their traffic [3, 14]. These systems, however, enable comparison only among ISPs from which the customer already buys service. Our goal is to let customers compare arbitrary ISPs to guide their purchasing decisions in the first place.

There exist a few listings to compare arbitrary ISPs, but most of these are focused on broadband and dial-up ISPs [12, 17]. For backbone ISP comparison, which is our focus, we are aware of only Keynote [20]. It measures latency and loss rate for paths internal to ISPs and paths between pairs of ISPs. For these measurements, it co-locates nodes within some ISPs' points of presence (PoPs) and measures the paths between the nodes.

Keynote's approach for comparing ISPs has several limitations. First, because it requires active cooperation from ISPs to place nodes inside their PoPs, Keynote's coverage of an ISP's network is poor (Section 6.2). This node placement strategy is also more vulnerable to ISPs that wish to game the measurements as it is easier to separate probe and actual traffic. Second, like SLAs, it does not characterize the end-to-end path. Third, Keynote's probes are addressed to its measurement nodes and not actual destinations. Because Internet routing is destination-based, the performance experienced by destination-bound traffic may differ from measurement-node-bound traffic.

2.2 Other Work on Network Measurement

We draw heavily on works that infer ISP topologies [15, 19, 37]. We use many existing techniques, e.g., DNS-based mapping of IP addresses to geographical locations. We also extend some existing techniques. For instance, our set covering-packing abstraction for reducing probe traffic is a more general and flexible formulation than the heuristics used in Rocketfuel [37].

We also draw on systems that construct network-wide information sources to estimate the performance of various paths [13, 27, 31]. We share with them the challenge of controlling probing overhead. However, existing systems cannot be used for ISP comparison because

they reduce overhead by abstracting away details crucial for such comparison. For instance, iPlane [27] views the network as a collection of links; to estimate performance between two hosts, it firsts estimates the series of links along the path and then bases path performance on link performances. In this procedure, errors in path estimates lead to incorrect performance estimates and the impact of destination-based routing is ignored. We believe that capturing these effects accurately is necessary for reliable ISP comparison, which led us to develop different methods for reducing probing overhead.

Our key contributions, however, lie not in developing new topology or performance measurement techniques but in formulating the problem of ISP comparison and building a practical system for it. Previously, ISP topology inference work has not measured performance and performance measurement work has not compared ISPs.

3 Goals and Approach

Our goal is to characterize how well traffic handed to an ISP is delivered to the destination. This traffic includes what is sent by the ISP's customers to various destinations and what is sent by the ISP's neighbors to the customers. Rather than performing a coarse characterization that ranks ISPs without regard to the properties of the traffic, we want to enable consumers to compare and decide which ISP is better for their specific traffic. Thus, we must uncover detailed differences in ISPs' performance along dimensions of interest to consumers. These include geographical location of end points, types of sources and destinations (e.g., content provider versus end users), and stability of performance. For instance, a content provider in Los Angeles should be able to determine which ISP delivers best performance to users in East Asia.

As the measure of performance, this paper focuses on latency, i.e., the time packets take to reach their destination after they are handed to the ISP. Latency has a first order impact on the performance of many applications (Section 6.4). Ongoing work is extending our system to other measures such as loss rate and available bandwidth.

3.1 Requirements for a System to Compare ISPs

Inspired by benchmarks for file systems and databases [9, 10, 40], we require our system to quantify performance in a way that is relevant to consumers, enable fair comparison, and help ISPs make informed optimizations. We describe these requirements in more detail below.

1. Relevant to applications and ISPs' customers The primary requirement is that our inferences be directly relevant to consumers; this has three implications:

- i)* Measure the performance of the entire path from where the traffic enters the ISP to the destination, not just the internal component. One might argue that the external component be discounted because the ISP does not

control it directly. We argue for its inclusion because the performance of the entire path is what matters to applications. Additionally, the ISPs can influence the quality of the entire path, through routing and peering decisions.

- ii)* Reflect the experience of application traffic. This means that we use traffic addressed to destinations of interest and not extrapolate application performance from the performance of the underlying links. The latter may not reflect application experience because of possible routing issues. The desire to reflect application experience also suggests that we passively measure the performance of real traffic, but we defer this to future work.

- iii)* Along with a long-term, average view, capture performance over short intervals. Short-term views provide their own utility because they enable wide-area applications to make short-term adjustments, inform customers of the variance in an ISP's performance, and provide information on how an ISP performs during periods of increased importance to the customers (e.g., day versus night). Based on the timescales of routing dynamics in the Internet [22], we target a period of 15 minutes to capture a snapshot of an ISP's performance. As discussed later, this places a significant demand on our system.

2. Fair comparison across ISPs Our measures of ISP performance should account for inherent differences in ISP networks, such as their size and geographic presence. For instance, it is unfair to compare the average time that traffic spends in two ISP networks when one is international and the other is regional.

To account for differences among ISPs, instead of viewing them as networks of routers and links, we view them as networks that connect cities by inferring the locations of their routers. Combined with inferences about the geographical location of destination networks, it lets us normalize results based on the geographical distance between the end points. It also enables customers to focus exclusively on ISPs that serve their needs. For instance, some customers may be interested only in paths between Los Angeles and Europe.

3. Helpful to ISPs Our system should also help ISPs better understand their own performance. They should be able to tell, for instance, whether performance issues that customers experience stem from problems inside their own network or outside their network, and whether performance is particularly bad from certain cities. The resolutions of these problems are different in each case.

3.2 Architecture of Netdiff

Building a system to compare ISPs is challenging because ISPs are embedded in an inter-network of many other ISPs. Unlike file and database systems, we cannot bring an ISP network to a laboratory and construct a measurement harness around it [9, 10, 40]. Instead the

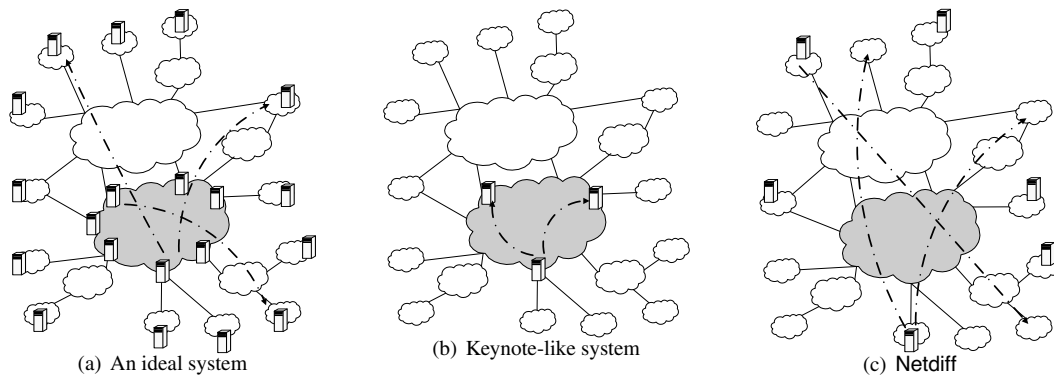


Figure 1: Three different architectures for measuring ISP performance. The shaded cloud represents the target ISP network, and the boxes represent measurement nodes.

ISP network must be measured *in situ*, but deploying such a harness can be difficult.

To understand how one might build a system to characterize an ISP network, consider an ideal system of Figure 1(a). This system has measurement nodes inside each PoP and each destination network because we want to measure performance from PoPs where the ISP is handed the traffic to destinations. With this system, the measurement task is straightforward but the deployment barrier is high. It requires thousands of measurements nodes to measure paths to even a fraction of destinations networks. It also requires significant cooperation from ISPs to place nodes inside their PoPs. Many ISPs may be unwilling or unable to provide this level of cooperation.

Keynote circumvents these problems by placing nodes inside only a few PoPs of cooperative ISPs and measuring only paths between those nodes. Figure 1(b) shows our understanding of Keynote’s system. It, however, has the previously mentioned limitations: inability to measure paths to destinations and poor coverage (Section 6.2).

Netdiff, illustrated in Figure 1(c), approximates the capability of the ideal system but is easy to deploy. We place measurement nodes inside edge networks. To provide good coverage with a modest number of nodes, we use single-ended, traceroute-like probes to destinations. From these probes, we infer the performance of the relevant subpaths from entry into the ISP to the destination. We identify subpaths by first inferring the ISP’s topology.

Another advantage of Netdiff over Keynote is robustness to ISPs that wish to game the measurements. It is harder to separate probe and actual traffic from hosts inside edge networks. Masking techniques can further raise the bar for this separation [30].

4 Measurement and Analysis Techniques

While our architecture is easy to deploy, engineering it is challenging. We must: *i*) aggressively reduce the number of probes; *ii*) extract performance information about the subpath of interest from the end-to-end probes; and

iii) make our inferences robust to measurement noise. We consider each challenge in successive subsections.

4.1 Reducing Probing Requirement

Because we place measurement nodes inside edge networks, we must limit probing overhead to control the bandwidth cost for host networks and to not overload their access links. To understand the need for limiting the probing requirement, assume that there are 200 measurement nodes and 250 K destination IP prefixes, the current size of the BGP routing table. Also assume that there are twenty hops in a path and it takes a 100-byte probe to measure to each hop. Then, if we want to measure an ISP within 15 minutes probing from all nodes to all prefixes requires a prohibitive 1 Gbps of probing traffic. We use the following methods to reduce this overhead.

Use BGP atoms Instead of using IP prefixes as destinations, we use BGP atoms. Atoms are groups of prefixes whose paths and routing dynamics are similar [1, 8]. It is not necessary that *all* atoms, as inferred using BGP tables, are “atomic.” But using atoms instead of prefixes significantly reduces the number of destinations and presents a worthwhile trade-off [27].

Select probes based on recent routing history Probes from a measurement node to many destinations do not traverse the ISP of interest and thus are not useful for measuring that ISP. We use a view of routing paths from the measurement node to restrict probing to paths that traverse the ISP. Before a node is used for measurements, it collects its view of routing to all the destinations. After that, this view is continuously refreshed using low-rate background probing.

Eliminate redundancies The set of all possible probes include many redundancies. For example, if probes from two nodes enter the ISP at the same place, only one is required. Similarly, for paths internal to an ISP, a probe that traverses three PoPs provides information about performance between three pairs of cities; other probes that traverse individual pairs are redundant. Eliminating such

redundancies can lower the probing overhead and also balance load across nodes.

The redundancy elimination problem is the following. Suppose we know the path of each possible probe. Of these, we want to select a subset such that: *i*) each ISP city to destination network is probed; and *ii*) each internal path between two cities is probed; *iii*) probing load at a measurement node does not exceed a threshold.

The problem above is an instance of the set covering-packing problem [21]: given multiple sets over a universe of elements, pick a subset of input sets such that each element is included at least a given number of times (covering constraint) and no element is included more than a given number of times (packing constraint). In our case, the input sets are probes, and the elements are paths to destinations, internal paths, and measurement nodes. Probes typically contain all three element types. This formulation of redundancy elimination is more general than the three heuristics used in Rocketfuel [37]. It is also more flexible in that it can systematically assign load based on a node's ability.

The set covering-packing problem is NP-hard, but greedy heuristics are known to yield good solutions [21]. After casting our input data into a set covering-packing problem, we implement one such greedy heuristic. Probes are added to the measurement set until all elements are covered at least once. At each step, the probe that covers the most as yet uncovered elements is added.

4.2 Recovering Network Topology

To extract paths of interest, we need to discover where an ISP begins and ends in the measured path and map its IP addresses to their respective locations. We use existing methods based on DNS names and majority voting for this task [37, 43]. For instance, the name `sl-gw12-sea-4-0-0.sprintlink.net` corresponds to a router belonging to Sprint in Seattle. We extend *undns* [37] with new naming rules to increase the number of names that are successfully deciphered.

We infer the location of destination networks using the commercial geolocation database from MaxMind [29]. This database is compiled from websites that ask users for their location. MaxMind claims that its accuracy is 99% in determining the country of an IP address. Within a country its stated accuracy varies and is stated to be 80% within the USA. Its predictions are used only if they pass our tests (see below).

4.3 Dealing with Errors and Noise in Data

There are several sources of noise and errors in our data. To make our inferences robust, we design tests to detect sources of erroneous data and filter them appropriately.

IP to ISP mapping An IP address may be incorrectly mapped to an ISP, e.g., due to an erroneous DNS

name. We check for such errors by observing the gathered traceroute data. The IPs belonging to the same ISP must appear consecutively, e.g., they should not be separated by an IP that belongs to another ISP. Transient routing problems can cause such an anomaly as well, but if we observe such an anomaly across many traceroutes, we can conclude that the ISP of the intervening IP has been incorrectly assigned.

Router IP to location mapping An IP address may be assigned an incorrect location, e.g., again due to an erroneous DNS name. We check for such errors using two tests. First, the traceroute for an ISP should not exit and then re-enter a city. As before, some of these anomalies arise because of transient routing issues; however, persistent issues indicate incorrect location mapping. The location mapping of an IP that is frequently sandwiched between two IPs belonging to a single different location is likely incorrect.

Second, we run a speed of light test among neighbors to detect erroneous mappings. The differences in the round trip latency observed to neighboring IPs should be more than the minimum time it takes for light to travel between the locations of the IPs. The latter time is calculated using the geographical coordinates assigned to particular locations and the speed of light in fiber. Thus, this test detects problems in assignment of geographic coordinates as well. If an IP fails this test for a majority of its neighbors, we conclude its location to be incorrect. Because of asymmetric routing, this test may fail for individual pairs even when the underlying data is correct.

Depending on the ISP, only 0.2-1.1% of traceroutes fail one of the two tests above. Deleting the mapping of a handful of IPs resolves the anomalies.

Geolocation for destination networks To detect errors in the geolocation database, we again use a test based on speed of light. Using traceroutes to the destination, we compare: *i*) the difference in the round trip latency between the destination and intermediate IPs with known locations; and *ii*) the minimum time for light to travel that path. Destinations for which the former is often less than the latter are deemed as having incorrect locations.

Path asymmetry Because we infer path latency using single-ended measurements, we must guard against our inferences being confused by significant asymmetries in forward and reverse paths. We discard traceroutes for which forward and reverse path length to an IP of interest differs by more than three. The reverse path length is inferred using the remaining TTL in the probe response. In Section 6.3, we show that this technique allows us to obtain reasonably accurate latency estimates by filtering out significantly asymmetries.

Overloaded node or local links Finally, in initial testing we found that our latency estimates were being corrupted by overloaded probing nodes or overloaded links

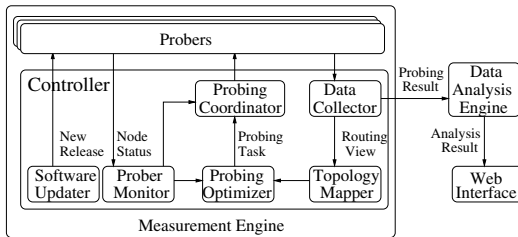


Figure 2: Main functional modules of Netdiff.

before the path entered the ISP. We now detect these events by observing the variance in the round trip time from the node to where the probe enters the ISP and discard data from nodes with high variance. A threshold of 40 ms worked well in our experiments. We demonstrate the effectiveness of this technique in Section 6.3.

5 Implementation of Netdiff

The implementation of Netdiff consists of measurement and data analysis engines and a Web interface, as illustrated in Figure 2. We describe each component below.

5.1 Measurement Engine

Netdiff divides the measurement process into cycles, measures one ISP per cycle, and iterates through the list of ISPs. This functionality is spread across a centralized *controller* and multiple *probers*. At the start of a cycle, the controller sends a list of probe destinations to each prober. When the probers finish, the probing results are recovered and the next cycle begins.

The key challenge in executing the process above is to start new cycles frequently, which is limited by two factors. The first one is the time for which the system not probing and is instead conducting some other support task. Done naively, even the simple task of transferring a single file to probers can run into tens of minutes. The second factor is the need to synchronize all probers at the beginning of each cycle because probers take different amounts of time to finish probing in a cycle. We control these factors through aggressive parallelization and termination of slow tasks.

Probers Probers measure path quality and maintain a fresh routing view to all BGP atoms. Upon receiving the list of destinations for a cycle, probers measure the paths to those destinations at the maximum allowed rate. Probing for the routing view is done at a low rate and spread over a day due to the large number ($\sim 55K$) of atoms.

We use a customized version of traceroute for probing. Probes are constructed to maximize the chance that all probe packets for a destination take the same router-level path in the network [7]. We probe multiple hops of a path in parallel and probe multiple destinations in parallel, subject to the maximum rate of 60 Kbps.

Controller The controller has six submodules:

1. *Topology mapper* recovers ISP topologies from routing views (Section 4.2). It evolves inferred ISP topologies with time by expiring routers and links that have not been observed for a week. It also refreshes DNS names of IP addresses once a week. To enable analysis of old data, we save the current view of ISP topologies once a day.

We continuously monitor the quality of the current routing and topological views. The monitoring script looks for indicators such as the number of IP addresses in an ISP, the number of IP addresses not resolved by *undns*, and the number of anomalous traceroutes. This script guides if any action, e.g., adding new *undns* rules, is needed. It also brings major topological changes, such as ISP mergers, to our attention.

2. *Probing optimizer* generates a list of probe destinations for each prober, based on the target ISP, the list of currently live probers, and routing views. It uses the redundancy elimination algorithm described earlier.

3. *Probing coordinator* drives the measurement cycles. At the beginning of each cycle, it transfers a list of probe destinations to each prober. To guard against slow transfers, it uses a customized version of parallel *scp* which ensures that each file is either completely copied within a minute or not copied at all. When it finds from the prober monitor (below) that all probers have finished, it moves to the next cycle. To prevent a measurement cycle from being blocked by a slow prober, it terminates probing activity after 15 minutes. Depending on the target ISP, 88-100% of the probers are able to finish their tasks in time. If not finished before, after this time limit, all probers are ready for the next cycle. Probers can usually measure roughly 9K paths per cycle.

4. *Prober monitor* periodically polls each prober to determine its current status – dead, busy, or idle.

5. *Data collector* copies the routing view and probing results from the probers to the controller.

6. *Software updater* ensures that all probers have up-to-date probing software and configuration files.

The controller performs almost all of the tasks in parallel. In our earlier implementation that performed many tasks sequentially, we found that much time was spent not probing because any delay (e.g., in data collection) slowed the entire chain. In our current version, the only tasks that are not performed in parallel with probing are transferring destination lists and checking prober status towards the end of a cycle. These tasks take roughly 2 minutes, which lets us start a new cycle every 17 minutes, of which 15 are spent probing.

5.2 Data Analysis Engine

Netdiff converts raw measurements into results that can be used to compare ISPs in three steps. The first step is to extract information about the paths of interest, i.e., those

between pairs of cities of an ISP and between an ISP's city and a destination atom. Consider an example path, observed while measuring AT&T's network:

```
Traceroute 128.112.139.71 -> 63.118.7.16 (atom 12322)
 1      *           0.542ms
 2  12.123.219.133  6.708ms   (AT&T New York)
 3  12.123.0.101    32.232ms  (AT&T Boston)
 4  63.118.7.1      36.345ms  (atom 12322)
```

The topology information maps the second and third hops to AT&T New York and Boston. From this, we can extract a latency sample for the path *NewYork*→*Boston*. While the traceroute does not reach the exact destination IP, it reaches the same atom. So we can extract latency samples for two destination paths, *NewYork*→*Atom12322* and *Boston*→*Atom12322*. In this step, we remove latency samples that are impacted by path asymmetry or overload (Section 4.3).

In the second step, the remaining latency samples of a path, which may come from different sources, are aggregated into one estimate for the path in that cycle. There are many methods to accomplish this, e.g., arithmetic mean, geometric mean, median, etc. We want a method that is robust to both small and large outliers. Following Roughan and Spatscheck [33], we use the median to aggregate samples from the same source and the arithmetic mean to aggregate across sources.

The final step produces an ISP-wide performance measure across a set of paths and cycles. We use two measures, *stretch* and *spread*, which we believe to be of broad interest; users can also access the output of the second step (see below) to perform this aggregation as they desire. We first aggregate across cycles and produce two values per path. The *stretch* of a path is the additional latency compared to that of a hypothetical direct fiber link between its two end points. For instance, the stretch is 10 ms if the latency is 40 ms for a path between cities whose direct-link distance is 30 ms. As representative latency of a path, we use the trimmed mean, which is the mean of the latency estimates between the 10th and 90th percentile. Using *stretch* instead of absolute latency enables aggregation across paths with different distances between their end-points. The *spread* of a path captures latency variation, using the difference between the 90th and 10th percentile latency samples. We then aggregate across paths using the arithmetic mean of *stretch* and *spread* of individual paths.

5.3 Web Interface

The Netdiff data are publicly available at <http://www.netdiff.org/> in two forms. Users who desire a detailed analysis based on their workload can download and process per-path latency estimates. Additionally, we have built an interface to support queries that are likely to be common. Feedback from a real consumer (Section 6.5) suggests that users will often be interested in aggregation based

on geography and observing the historical performance of paths of interest. Our interface supports these queries.

Our Web interface makes the data available to users a few hours after the measurements are conducted. We wanted to make a cycle's data available right after it ends, but this task was complicated by the fact that sometimes the results from a probe cannot be retrieved immediately after the cycle. To avoid the need for running the analysis repeatedly as new results trickle in, we wait for all the result files to be retrieved. Maximum waiting time is set to six hours. In the future, we plan to make our analysis engine operate incrementally on new results, which would enable us to provide results in almost real-time.

6 System Evaluation

We have instantiated Netdiff on roughly 700 PlanetLab nodes, which are spread across roughly 300 sites. It has been operational since February 2007. In this section, we use this instantiation to evaluate its design. The next section compares ISPs.

We consider the following questions in subsequent subsections. *i*) By how much do our optimizations reduce the probing requirement? *ii*) What is the extent of coverage that Netdiff achieves for ISPs with the current set of nodes? *iii*) Are Netdiff's path latency estimates reliable? *iv*) How do its latency estimates relate to application performance? *v*) Are consumers likely to find Netdiff useful?

Table 1 lists the ISPs that Netdiff currently measures. All of these are major backbone ISPs with different primary geographic regions of operation. While which ISPs are "tier 1" is always open to debate, our list is a superset of the ten tier 1 ISPs as per one source [38]. The second column shows the number of cities in which the ISP has a PoP according to our data. Some ISP cities may be missing. However, past work shows that, even with far fewer than 300 sites, almost all cities are captured using measurements that are similar to ours [23, 37].

6.1 Probing Requirement

The third column of Table 1 shows the average number of paths that Netdiff measures for each ISP. It shows in parenthesis the multiplicative reduction brought by our set covering-packing based optimization. We see that the optimization reduces probes by roughly an order of magnitude. Of the two other probe reduction techniques, using BGP atoms brings roughly a four-fold reduction, from 250K IP prefixes to 55K atoms. Using routing history to select probes brings another order of magnitude reduction, from 16,500K ($300 \times 55K$) probes to fewer than 1,200K on average. Because these reductions multiply, the three methods together reduce probing requirement by a factor of 400.

Observe from the table that the number of paths probed for an ISP is not simply a function of the number of cities.

					Keynote
ISP	# cities	# probes (K) (reduction factor)	# destination paths (% of total)	# internal paths (% of total)	# internal paths (% of Netdiff)
AOL Transit	27	5 (5.5)	658 (0.05)	230 (32.8)	n/a
AT&T	113	86 (13.5)	13364 (0.24)	838 (6.6)	72 (8.6)
AboveNet	20	40 (7.6)	17258 (1.73)	277 (72.8)	n/a
British Telecom	32	11 (13.7)	4898 (0.31)	440 (44.3)	2 (0.5)
Broadwing	23	28 (9.5)	7655 (0.67)	149 (29.3)	n/a
Cogent	72	161 (10.2)	42620 (1.18)	1799 (35.2)	12 (0.7)
Deutsche Telekom	67	29 (7.7)	2266 (0.07)	129 (2.9)	0 (0.0)
France Telecom	25	31 (12.1)	6092 (0.49)	229 (38.1)	n/a
Global Crossing	60	143 (9.0)	19082 (0.64)	689 (19.4)	n/a
Level3	62	249 (12.7)	70907 (2.29)	1513 (40.0)	30 (2.0)
NTT/Verio	46	98 (8.4)	28943 (1.26)	535 (25.8)	6 (1.1)
Qwest	52	112 (10.9)	20270 (0.78)	696 (26.2)	30 (4.3)
Savvis	41	56 (8.2)	10012 (0.49)	511 (31.1)	20 (3.9)
Sprint	55	136 (13.1)	36366 (1.32)	1208 (40.7)	20 (1.7)
Tiscali	36	30 (8.8)	5483 (0.30)	325 (25.7)	0 (0.0)
VSNL (Teleglobe)	43	30 (14.4)	5500 (0.26)	542 (30.0)	6 (1.1)
Verizon	161	120 (11.5)	20507 (0.26)	2098 (8.1)	306 (14.6)
XO	47	7 (23.5)	1415 (0.06)	522 (24.1)	2 (0.4)

Table 1: *Backbone ISPs that Netdiff currently measures. The contents of the columns are explained in the text.*

It depends on several factors, including the location of the probes and the structure of the ISP network. It is higher for MPLS-based networks such as Level3. With MPLS, probes usually observe only the ingress and egress cities of the ISP network because MPLS hides the intermediate cities that are traversed. Without MPLS, they observe other cities along the path as well. Thus, individual probes provide more information about the ISP network, which helps to reduce the number of probes.

6.2 Path Coverage

Next, we study the number of paths Netdiff measures for an ISP from the current set of sites. We consider both destination paths, which begin at the ISP and end at destination networks, and paths internal to the ISP.

The fourth column in Table 1 shows the coverage for destination paths. The percentage is computed based on the total number of possible destination paths and excludes paths that are filtered due to noise and errors. We can measure thousands of paths for almost all ISPs, which represents 0.05-2.29% of all paths. We find that filtering significantly lowers coverage in our current implementation. We are investigating if probes can be assigned to probers such that they are less likely to be filtered, e.g., based on reverse path lengths.

We believe, however, that even the current coverage level of Netdiff is sufficient to derive a reasonable view of an ISP's performance. If our measured paths are not heavily biased, we measure enough paths to obtain representative measures of performance. A similar assumption is used by opinion polls that estimate voting results by sampling a minuscule percentage of the population.

While the absence of bias in our measurements is hard to determine with certainty, we conduct several sanity tests to evaluate it. Section 7.3 presents one such test.

The next column shows the coverage for internal paths. It is significantly higher than that of destination paths and varies between 2.9-72.8%. Like the probing requirement, coverage depends on several factors including network topology. For instance, it is low for AT&T because AT&T has a hub-and-spoke topology with many small cities that connect to big cities [37]. Covering paths that begin at smaller cities is harder due to the lack of probers there.

For comparison, we estimate the current coverage of Keynote. Because Keynote does not measure destination paths, we consider only internal paths. It claims to have 1,200 measurement sites worldwide. Based on the list at http://www.keynote.com/support/technical_information/agent_location.html, the last column of Table 1 shows the number of internal paths that it can measure by sending probes between its nodes. The value is 0 for ISPs in which it has only one agent and "n/a" for ISPs in which it has none. Even where Keynote has multiple nodes, its coverage is one to two orders of magnitude less than Netdiff. This difference stems from both our single-ended measurement methodology and that we do not require active cooperation from ISPs. Keynote could in principle start using single-ended measurements, but it would then end up adopting our architecture and would need to address challenges that we tackle in this paper.

6.3 Accuracy of Path Latency Estimates

We now investigate the accuracy of Netdiff's latency measurements to sources of error that include: *i*) time to gen-

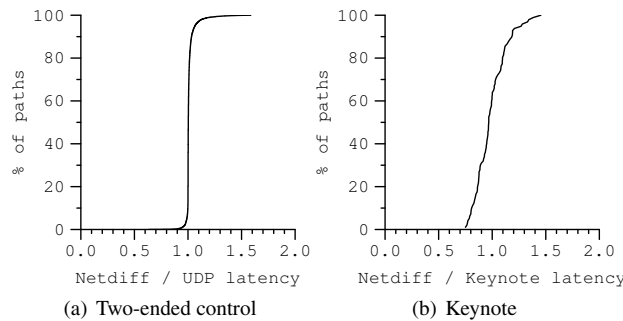


Figure 3: Comparison of Netdiff inferences with other methods. Graphs plot CDFs for common paths.

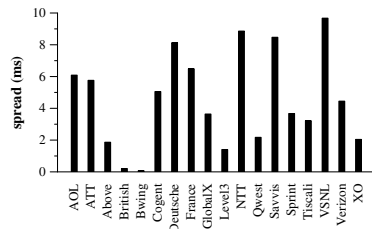


Figure 4: Average spread in latency measurements of the local path segment.

erate or forward ICMP responses; *ii*) path asymmetry; and *iii*) overloaded PlanetLab nodes or access links. Ideally, we would compare our inferences against an authoritative source. But since we do not have access to such a source, we use four evaluations that together suggest that our latency estimates are reasonably accurate.

Comparison with two-ended control For paths where we can control both ends, we compare Netdiff inferences to latency measured using the more accurate two-ended measurements. The paths we consider in this experiment are those between pairs of PlanetLab nodes. While most such paths traverse the research and education networks, many do not because of PlanetLab sites that are located outside of universities; our results are similar for both kinds of paths. We compare path latencies as measured using Netdiff and using contemporaneous one-way UDP probes in both directions. Because it relies on ICMP responses in the reverse direction, Netdiff will perceive a higher latency than UDP probes if ICMP packets are commonly delayed in transit.

Figure 3(a) shows the relative difference in the latency measurements of UDP probes and Netdiff. We see that the two methods almost always agree. High relative difference in the tail corresponds to very short paths.

Latency variation on the local path segment We now measure the extent of variation in measured latency along the path segment before entering the ISP. This variation can stem from overloaded PlanetLab nodes or overloaded links along this segment. It helps us estimate the impact of local characteristics on Netdiff's measurements.

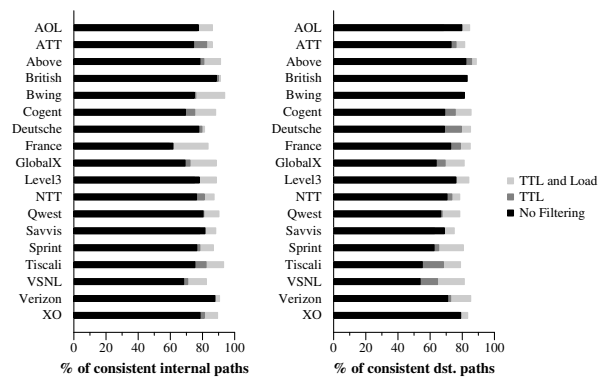


Figure 5: Consistency of inferences across sources.

This estimate, however, represents an upper bound on the impact, because Netdiff subtracts the local latency while predicting the latency of the path of interest.

Figure 4 shows that the average latency spread (Section 5.2) for local path segments is typically low, with a maximum value of around 10 ms.

Consistency across sources We now test the consistency of inferences for paths for which we obtain more than one measurement in the same cycle from different nodes. If our estimates are not confused by reverse path asymmetry or local load, these inferences should roughly agree. We consider multiple inferences of a path to be consistent if *all* of them lie within 10 ms. Given that average latency of paths in our data is significantly higher and local variation is within 10 ms, this is a reasonably strong test of consistency.

Figure 5 shows the percentage of paths with consistent measurements. It also shows the percentage for the cases where no filtering is done and where only hop-length-based filtering is done. With both filtering methods, 80-90% of the paths are consistent. We do not expect a complete agreement because the path may be measured at different times. The high consistency level suggests that our inferences are not heavily impacted by noise.

Comparison with Keynote Finally, we compare our latency estimates with Keynote [20] for the paths that are common to both systems. We expect Keynote measurements to be accurate because of its simpler methodology. We compare measurements conducted by the two systems on the same day, but they may not be exactly contemporaneous. If both systems reflect the latency of the underlying path, there should be a rough agreement between the two. On the other hand, if Netdiff estimates are impacted by any source of noise, e.g., ICMP response generation time, they would differ. Figure 3(b) plots the CDF of the average latency estimate of Netdiff divided by that of Keynote. We see that the two systems roughly agree. For 75% of the paths, the relative difference is within 20%. We also study absolute difference and find

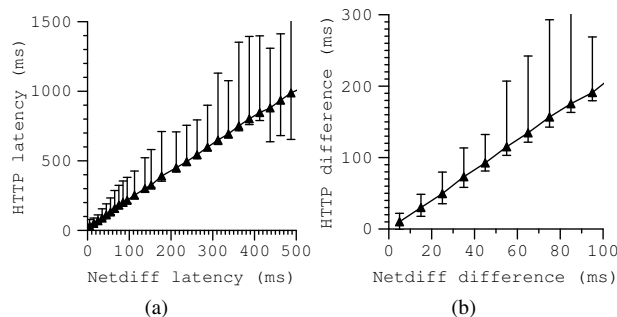


Figure 6: (a) *Relationship between the latency measured by Netdiff and the completion time for HTTP transaction.* (b) *The additional time it takes to complete an HTTP transaction as a function of the difference in path latency.* In both graphs, the lines connect the medians and the whiskers show the inter-quartile range.

that 90% of the paths are within 10 ms (not shown).

6.4 Correlation with Application Performance

We study the relationship between path latency measured by Netdiff and application performance. As the application, we consider HTTP transactions to Web servers and quantify performance as the time to complete the transaction. We start with 336K unique web servers visited by CoDeeN [11] users over a two-week period. We map each server to a BGP atom using its IP address. If multiple servers map to the same atom, we pick one randomly. This process yields a list of 8K web servers, each in a different atom. We measure path latency from all sources to these atoms using Netdiff. Contemporaneously, we download the default pages of these servers and log the time to complete each transaction.

Figure 6(a) shows the relationship between path latency measured by Netdiff and the time to complete an HTTP transaction. While HTTP transaction time is a complex function of not only path latency but also loss rate, server load, and page size, our results show that it is strongly correlated with latency estimates of Netdiff.

For pairs of paths to the same server, Figure 6(b) shows the additional time to complete an HTTP transaction along the longer path as a function of the latency difference measured using Netdiff. This further confirms that application performance would be poorer along paths that Netdiff predicts to have higher latency. Figure 6(b) also serves as a guideline for consumers of Netdiff analysis. For instance, if paths through two ISPs differ by 30 ms, the HTTP transaction times will typically differ by roughly 60 ms for small transfers that we use in this experiment and likely more for bigger transfers.

6.5 A Case Study on Usefulness to Customers

We gave early access to Netdiff inferences to operators of a large content provider and asked for their opinion. This

provider operates several data centers across the world and connects to many large ISPs. We summarize the operators' views here. This is not meant to be a scientific evaluation but highlights the strengths and limitations of Netdiff from the perspective of a real consumer.

The operators found the capabilities of Netdiff to be useful and novel (despite the fact that they are already customers of Keynote). They especially valued that they could determine the performance of an ISP from a data center city to various destinations. Netdiff lets them do this without signing new contracts with ISPs that they do not connect to and without changing their routing decisions. Of the many ways of observing Netdiff data, the most useful ones to them were being able to see performance based on geography and variations across time.

The operators also pointed out two capabilities that Netdiff does not currently possess but they would find useful. They wanted a close to current view of ISP performance, to aid performance troubleshooting. And they wanted to compare regional ISPs to big backbones for traffic that stays within a region. While these usage scenarios were not part of our original goal, they point at useful directions in which Netdiff can be extended.

7 ISP Comparison Results

We now present a series of results that compare ISPs in different ways. We begin with a comparison that can be considered as indicative of the overall quality of an ISP. Because this hides many differences that are of interest to individual customers and applications, we then compare ISPs in more detail by focusing on specific workloads. Our study is not exhaustive but highlights the kinds of detailed insights on ISP performance that Netdiff can provide. To our knowledge, such detailed information on ISPs' performance was not previously available.

The results below are for a month-long period between Feb. 13 - Mar. 14, 2007. In this period, Netdiff ran continuously without any unplanned or planned disturbance (e.g., for validation). Because of space constraints, we choose to focus on ISPs' performance averaged over the entire period rather than shorter-term variations.

Our results are of course limited to paths that we can measure using PlanetLab. Our sanity tests, however, show that the results are robust to the exact choice of paths. For instance, the results do not qualitatively change even if we discard half of the paths in our data. Section 7.3 describes another such test.

Figure 7 shows an example of the format we use to present results. The x -axis represents a performance measure. The y -axis shows the ISPs, sorted from best to worst. The whiskers represent 90% confidence interval around the average. For visual clarity, the x -axis range varies across graphs. To ease visual comparison, we divide ISPs into five roughly equal groups.

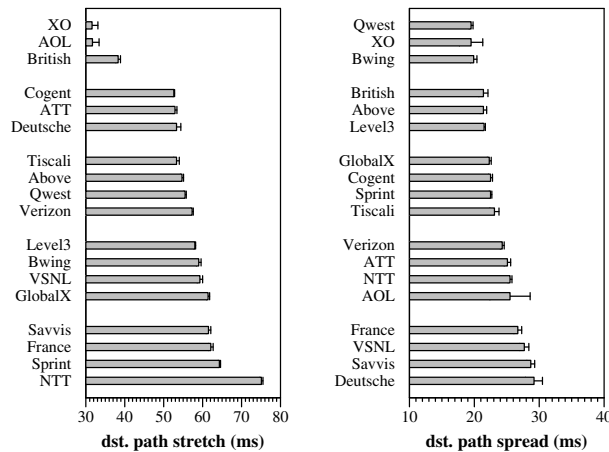


Figure 7: Stretch and spread of all destination paths.

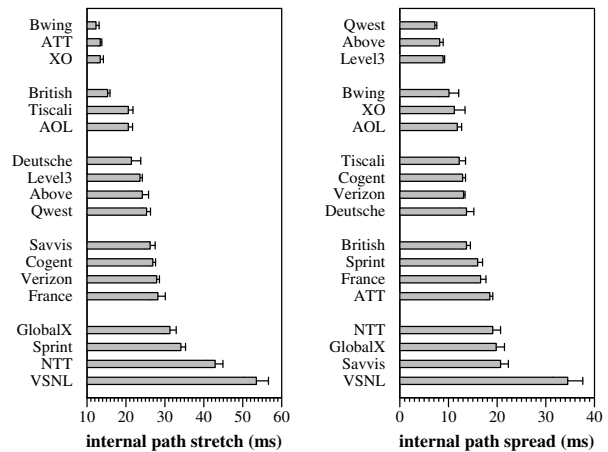


Figure 8: Stretch and spread of all internal paths.

Our results quantify the performance of each ISP, and whether the difference between two ISPs is significant is a question that customers must answer based on the needs of their applications. However, based on results in Section 6.3, we recommend that customers ignore performance differences of less than 10 ms between two ISPs.

7.1 Overall Comparison

Figure 7 shows the stretch and spread for all destination paths in our data. It is clear that the choice of ISP is important as the stretch offered by ISPs varies over a wide range. Further, the two measures order the ISPs differently. For instance, Qwest and Bwing have low spread but relatively high stretch. Thus, ISPs that offer the least stretch on average are not necessarily the same as those that also offer consistent path latency.

Figure 8 shows the stretch and spread for all internal paths. These measures provide a different ranking for ISPs. For instance, Bwing has relatively low internal path stretch but a high destination path stretch. Thus, good relative performance for internal paths does not necessarily translate to good relative performance for destination paths. An implication is that ISPs that offer better performance in their SLAs, which typically cover internal paths, may not offer better end-to-end performance.

Such analysis is helpful to not only consumers but also ISPs. For instance, an ISP can tell if problems behind poor performance of destination paths stem from inside its network or outside. For Bwing, for instance, the problems appears to be outside – it has good internal path performance – and changing interdomain neighbors and routing may improve performance.

Results aggregated across all paths hide many differences among ISPs that are relevant to consumers. The rest of this section compares ISPs in more detail.

7.2 Dependence on Distance

The first dimension that we study is the distance between the end points of paths. We divide paths into three groups based on the direct hypothetical link distance: *i) short*: less than 20 ms; *ii) medium*: 20-50 ms; and *iii) long*: more than 50 ms. Roughly, long paths are inter-continental, medium-length paths span a continent, and short paths are regional. Figure 9 shows the stretch for medium-length and long paths; for space constraints we omit short paths, which produce a different ordering for ISPs. The missing ISPs in a graph are those for which we have less than ten paths in that category. Many ISPs are missing in Figure 9(b), for long, internal paths, because few ISPs have inter-continental networks.

We see that stretch increases with path length and the relative performance of ISPs differs. While some ISPs are consistently good or bad per our measures, the relative quality of others varies. For instance, Bwing is in the top group for long destination paths but in the third group for medium-length paths. Performance for internal paths suggests that some ISPs are better at carrying traffic internally over shorter distances, while others are better over longer distances.

The six ISPs in the graph for long, internal paths have an inter-continental network. Interestingly, there appears to be little correlation with having an inter-continental network and the performance seen by their consumers for long, destination paths. Based on work that highlights that end-to-end paths can be long due to inter-ISP routing [34], we expected such ISPs to be better at delivering traffic to distant destinations because of potentially fewer inter-ISP transfers. But our results do not bear this out.

The generally higher internal path stretch for these six ISPs in Figure 8 – they all are in the bottom half – might tempt some to conclude that these ISPs are poor. But this is another instance of how judging an ISP only by its

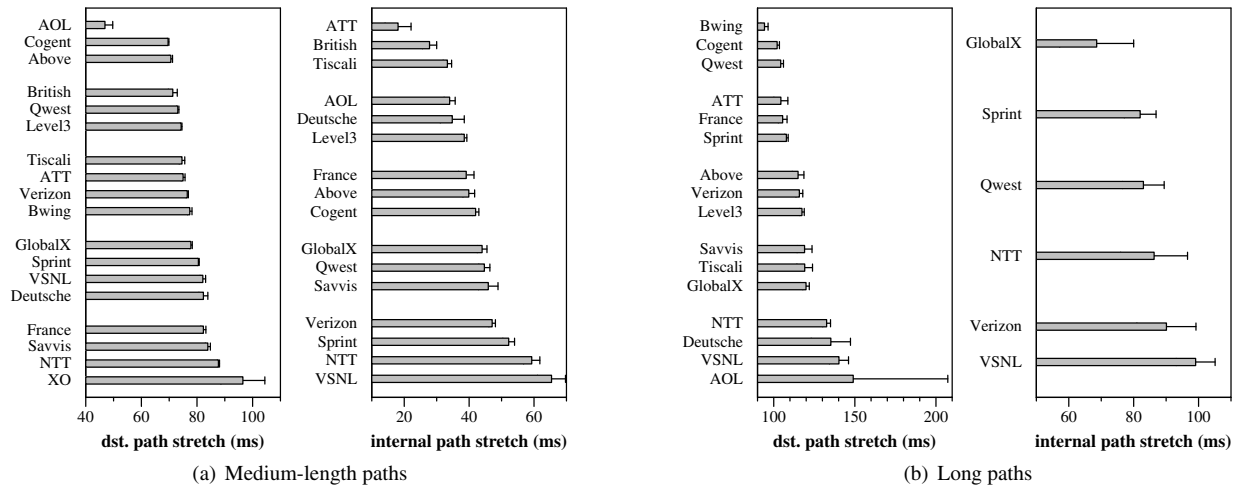


Figure 9: Stretch of medium-length (20-50ms) and long (over 50ms) destination and internal paths.

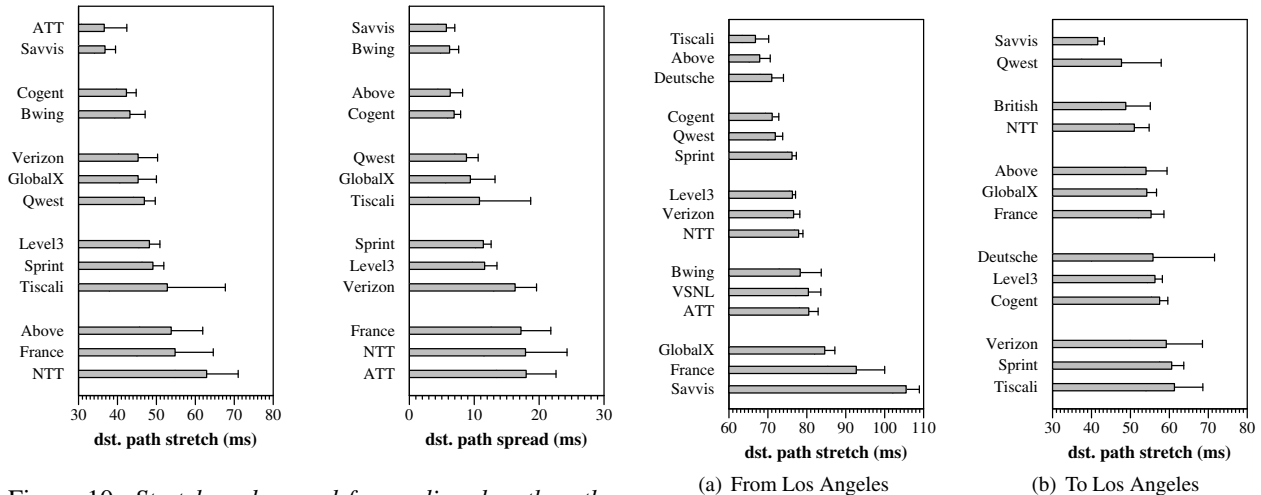


Figure 10: Stretch and spread for medium-length paths that originate and terminate in the USA.

internal paths or SLAs can be misleading. Our analysis shows that the higher internal stretch is simply reflective of their network size and not performance.

7.3 Dependence on Geographic Region

The second dimension that we study is dependence on geographic properties of the traffic. Many consumers will be interested in how an ISP delivers traffic to or from specific regions. We use two example scenarios to show that such consumers may make different choices than location-agnostic consumers.

First, consider consumers that are interested in only one country, perhaps because all of their important nodes reside there. Figure 10 plots the stretch and spread for medium-length paths that originate and terminate inside the USA. Based on the observations in the last section, we do not combine all path lengths. The relative ranking for this case is different than for all medium-length paths in Figure 9(a). For instance, Savvis, which was in

Figure 11: Stretch for medium-length destination paths that begin near Los Angeles and that terminate there.

the bottom group, is now in the top group. AboveNet moves in the opposite direction. Thus, consumer should make ISP choices based on whether their destinations are mostly domestic or international.

Second, consider consumers that are interested in paths originating or terminating in a specific geographic region. For this, we fold the cities in our data into metropolitan areas because different ISPs may use different city names for the same geographical region (e.g., San Jose versus Mountain View in California, USA). Starting with the list of all cities, we repeatedly select the city with most IPs and include in its metropolitan area all other cities that are within a 100-mile radius.

Figure 11 shows the results for the Los Angeles metropolitan area which is one of the biggest in our analysis. The graphs plot the stretch for medium-length paths originating or terminating near Los Angeles. Some of the

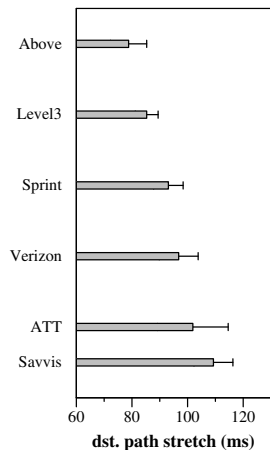


Figure 12: *Stretch for medium-length destination paths from Los Angeles to destinations that are common to ISPs.*

ISPs in this figure (e.g., Deutsche Telekom) may not offer a transit service along these paths. For them, the graphs capture performance that customers would experience if the ISP were to offer the service.

We see again that the order of ISPs is different from that for the case of all medium-length paths in Figure 9(a). For instance, Tiscali is significantly better for traffic originating near Los Angeles. Additionally, the relative performance of ISPs for traffic from Los Angeles is different from traffic to Los Angeles, likely because of early exit routing practice by which ISPs transfer traffic to the next ISPs at the closest inter-connection.

A sanity test To test if our results are sensitive to the set of paths that we study, we analyze the ranking for paths from Los Angeles but only to destinations that are common across ISPs. This tests, for instance, whether the differences among ISPs that we find is only because we measure different destinations through them. To ensure we have enough destinations for analysis, we consider only a subset of the ISPs with many common destinations. Figure 12 shows that the relative order of these ISPs is fairly consistent with Figure 11 even though the number of paths being considered is reduced to 6% for some of the ISPs. The only change is the inversion of the order of Sprint and Level3, which have similar stretch.

7.4 Dependence on Destinations

We now study dependence on properties of destinations. So far, we have considered paths to arbitrary destinations, which is more likely to be of interest to content providers. Other consumers may be interested in a specific types of destinations. For instance, a broadband provider may be interested in performance to popular websites. For this experiment, we consider the list of top 100 websites [4] as destinations. Figure 13 shows the ordering for ISPs for

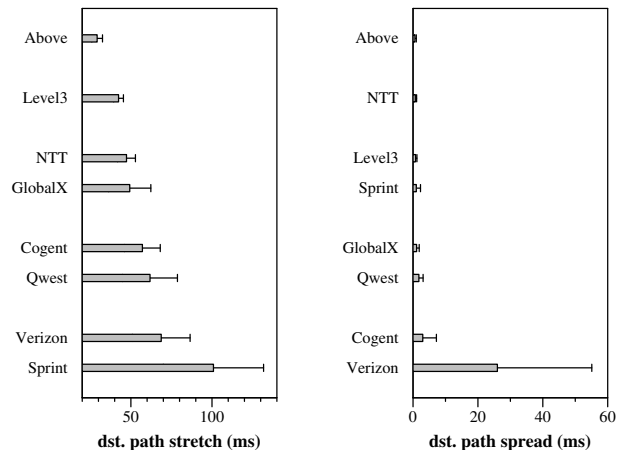


Figure 13: *Stretch and spread for medium-length destination paths to popular web sites.*

which we have enough paths to such destinations. Compared to Figure 9(a), the order changes significantly for some of the ISPs. For instance, for stretch, NTT moves up and Cogent and Qwest move down. We can also see that the performance to popular destinations is in general better than that to arbitrary destinations.

7.5 Summary

Our results show that ISPs differ in various ways, and they underscore the value of Netdiff because it enables customers and applications to pick the ISP that is best suited to deliver their traffic. As a concrete example, it can help a content provider in Los Angeles determine which ISP to use to send traffic to users in East Asia. For this, the provider can use our Web interface to determine the recent and historical performance of various ISPs in carrying traffic from Los Angeles to major cities in East Asia. If it is interested in specific destination networks, e.g., a major broadband service provider, it can use our inferences to make that determination as well. Once it has decided, perhaps after also accounting for cost, it can buy service from the chosen ISP and use it for traffic to East Asia.

8 Conclusions and Future Work

We built Netdiff, a system to uncover detailed performance differences among ISPs. Our work shows that it is possible to build such a system in a way that is easy to deploy, does not require active cooperation from ISPs, and has acceptable probing cost.

Our analysis revealed that the choice of the ISP can significantly impact application performance. But the relative ranking of ISPs depends on many factors, including the distance traveled by traffic and its geographic properties. It also revealed that application performance is not directly reflected in the quality of an ISP's internal

paths, the basis of typical SLAs today. Thus, the choice of ISP is a complicated decision that should be based on the properties of the workload. It is in this complex task that Netdiff helps consumers by enabling a detailed analysis of ISP performance.

This paper lays the foundation for our broader goal of objective and comprehensive comparison between ISPs. An obvious direction to extend Netdiff is to measure performance aspects beyond path latency. Single-ended, hop-by-hop measurement techniques for path loss, capacity, and bottleneck bandwidth [16, 18, 28] fit well within our current measurement framework. Another direction is to investigate why two ISPs perform differently. A starting point for this task is to correlate an ISP's performance to its internal network structure as well as to its peering and routing policies. Finally, Netdiff can be extended to compare behaviors beyond performance. For instance, we have started investigating if an ISP's "neutrality" can be measured by studying if they favor or disfavor certain types of traffic.

Acknowledgments We thank the NSDI reviewers and our shepherd, Krishna Gummadi, for their feedback. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

- [1] Y. Afek, O. Ben-Shalom, and A. Bremler-Barr. On the structure and application of BGP policy atoms. In *IMW*, Nov. 2002.
- [2] M. Afergan and J. Wroclawski. On the benefits and feasibility of incentive based routing infrastructure. In *PINS*, Sept. 2004.
- [3] A. Akella, S. Seshan, and A. Shaikh. Multihoming performance benefits: An experimental evaluation of practical enterprise strategies. In *USENIX Annual Technical Conference*, June 2004.
- [4] Alexa Web Wearch – top 500. http://www.alexa.com/site/ds/top-sites?ts_mode=lang&lang=en.
- [5] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, Oct. 2001.
- [6] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *HotNets*, Nov. 2004.
- [7] B. Augustin, X. Cuvelier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *IMC*, Oct. 2006.
- [8] A. Broido and kc claffy. Analysis of RouteViews BGP data: Policy atoms. In *Workshop on Network-Related Data Management*, May 2001.
- [9] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *USENIX Annual Technical Conference*, June 2000.
- [10] P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. *ACM TOCS*, 12(4), 1994.
- [11] CoDeeN: A CDN on PlanetLab. <http://codeen.cs.princeton.edu/>.
- [12] Shop, compare, and save on Internet service providers. <http://www.comparingisps.com>.
- [13] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM ToN*, 9(5), 2001.
- [14] G. Goddard and R. Vaughn. RouteScience's PathControl. <http://www.networkworld.com/reviews/2002/0415rev.html>, Apr. 2002.
- [15] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *INFOCOM*, Mar. 2000.
- [16] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *SIGCOMM*, Aug. 2004.
- [17] Cheapest Internet connections. <http://www.isp-connections.com/index.html>.
- [18] V. Jacobson. Pathchar. <ftp://ftp.ee.lbl.gov/pathchar>.
- [19] kc claffy, T. E. Monk, and D. McRobb. Internet tomography. In *Nature*, Jan. 1999.
- [20] Keynote. Internet Health Report. <http://www.internetpulse.net>.
- [21] S. G. Kolliopoulos and N. E. Young. Approximation algorithms for covering/packing integer programs. *Journal of Computer and System Sciences*, 71(4), 2005.
- [22] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. An experimental study of delayed Internet routing convergence. In *SIGCOMM*, Aug. 2000.
- [23] A. Lakhina, J. Byers, M. Crovella, and P. Xie. Sampling biases in IP topology measurements, Mar. 2003.
- [24] P. Laskowski and J. Chuang. Network monitors and contracting systems: Competition and innovation. In *SIGCOMM*, Sept. 2006.
- [25] J. Lewis. Cogent service. NANOG mailing list archive: <http://www.cctec.com/maillists/nanog/historical/0209/msg00684.html>, Sept. 2002.
- [26] D. Lundquist. Graphical displays of latency/packet loss for top Internet backbones. NANOG mailing list archive: <http://www.cctec.com/maillists/nanog/current/msg03940.html>, Sept. 2004.
- [27] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. An information plane for distributed services. In *OSDI*, Nov. 2006.
- [28] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP*, Oct. 2003.
- [29] MaxMind – GeoIP City geolocation IP address to city. <http://www.maxmind.com/app/city>.
- [30] M. Natu, A. Sethi, R. Hardy, and R. Gopaul. Design approaches for stealthy probing. Technical Report 2007/340, Dept. of CIS, Univ. of Delaware, Oct. 2007.
- [31] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale Internet measurement. *IEEE Communications*, 36(8), Aug. 1998.
- [32] S. Ratnasamy, S. Shenker, and S. McCanne. Towards an evolvable Internet architecture. In *SIGCOMM*, Aug. 2005.
- [33] M. Roughan and O. Spatscheck. What does the mean mean? In *Int'l Teletraffic Congress (ITC) 18*, 2003.
- [34] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *SIGCOMM*, Aug. 1999.
- [35] N. Semret. Cogent: Disruptive pricing or disruptive marketing. <http://web.invisiblehand.net/wp/IHN-WP-Cogent.html>.
- [36] SPEC – Standard Performance Evaluation Corporation. <http://www.spec.org>.
- [37] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM ToN*, 12(1), 2004.
- [38] Tier 1 network. http://en.wikipedia.org/wiki/Tier_1_carrier.
- [39] TPC – Transaction Processing Performance Council. <http://www.tpc.org>.
- [40] T. K. Tsai, R. K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Symp. on Fault-Tolerant Computing*, June 1996.
- [41] X. Yang, D. Clark, and A. Berger. NIRA: A new routing architecture. *IEEE/ACM ToN*, 15(4), 2007.
- [42] M. Zeier. Genuity – any good? NANOG mailing list archive: <http://www.cctec.com/maillists/nanog/historical/0204/msg00318.html>, Apr. 2002.
- [43] M. Zhang, Y. Ruan, V. S. Pai, and J. Rexford. How DNS misnaming distorts Internet topology mapping. In *USENIX Annual Technical Conference*, June 2006.

Effective Diagnosis of Routing Disruptions from End Systems

Ying Zhang
University of Michigan

Z. Morley Mao
University of Michigan

Ming Zhang
Microsoft Research

Abstract

Internet routing events are known to introduce severe disruption to applications. So far effective diagnosis of routing events has relied on proprietary ISP data feeds, resulting in limited ISP-centric views not easily accessible by customers or other ISPs. In this work, we propose a novel approach to diagnosing significant routing events associated with any large networks from the perspective of end systems. Our approach is based on scalable, collaborative probing launched from end systems and does not require proprietary data from ISPs. Using a greedy scheme for event correlation and cause inference, we can diagnose both interdomain and intradomain routing events. Unlike existing methods based on passive route monitoring, our approach can also measure the impact of routing events on end-to-end network performance. We demonstrate the effectiveness of our approach by studying five large ISPs over four months. We validate its accuracy by comparing with the existing ISP-centric method and also with events reported on NANOG mailing lists. Our work is the first to scalably and accurately diagnose routing events associated with large networks entirely from end systems.

1 Introduction

The end-to-end performance of distributed applications and network services is known to be susceptible to routing disruptions in ISP networks. Recent work showed routing disruptions often lead to periods of significant packet drops, high latencies, and even temporary reachability loss [1, 2, 3, 4]. The ability to pinpoint the network responsible for observed routing disruptions is critical for network operators to quickly identify the problem cause and mitigate potential impact on customers. In response, operators may tune their network configurations or notify other ISPs based on the inferred origin location of the routing disruption: internal networks, border routers, or remote networks. They may also find alternate routes or inform impacted customers about destinations expected to experience degraded performance.

From the perspective of end users, the ability to diagnose routing disruptions also provides insight into the reliability of ISP networks and ways to improve the network infrastructure as a whole. Knowing which ISPs should be held accountable for which routing disruptions helps customers assess the compliance of their service-

level agreements (SLAs) and moreover provides strong incentives for ISPs to enhance their service quality.

Past work on diagnosing routing events has relied on routing feeds from each ISP. These techniques have proven to be effective in pinpointing routing events across multiple ISPs [5] or specific to a particular ISP [6]. However, given that most ISPs are reluctant about revealing details of their networks, they normally keep their routing feeds publicly inaccessible. Today, the largest public routing data repositories, RouteViews and RIPE, receive data from only around 154 ISPs [7, 8], in most cases with at most one feed from each AS. These feeds have been shown to be insufficient to localize routing events to a particular ISP [9]. As a result, customers are in the dark about whether their service providers meet their service agreements. Similarly, ISPs have limited ways to find out whether the problems experienced by their customers are caused by their neighbors or some remote networks. They usually have to rely on phone calls or emails [10] to perform troubleshooting.

Motivated by the above observations, we aim to develop new techniques for diagnosing routing events from end systems residing at the edge of the Internet. Our approach differs markedly from existing work on pinpointing routing events by relying only on probes launched from end-hosts and not requiring any ISP proprietary information. Using active probing on the data plane, our system can in fact more accurately measure the performance of actual forwarding paths rather than merely knowing the expected routes used based on routing advertisements. Furthermore, our techniques can be easily applied to many different ISPs without being restricted to any particular one. This is especially useful for diagnosing inter-domain routing events which often require cooperation among multiple ISPs. Our inference results can be made easily accessible to both customers and ISPs who need better visibility into other networks. This is also helpful for independent SLA monitoring and management of routing disruptions. In addition, end system probing can be used for both diagnosing and measuring the performance impact of routing events. It offers us a unique perspective to understand the impact of routing events on end-to-end network performance.

In this paper, we consider the problem of diagnosing routing events for any given ISP based on end system probing. Realizing that identifying the root cause of

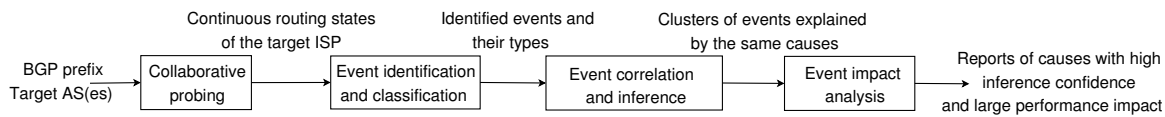


Figure 1: System Architecture

routing events is intrinsically difficult as illustrated by Teixeira and Rexford [9], we focus on explaining routing events that the ISP should be held accountable for and can directly address, *e.g.*, internal routing changes and peering session failures. In essence, we try to tackle the similar problem specified by Wu *et al.* [6] without using ISP’s proprietary routing feeds. Given that end systems do not have any direct visibility into the routing state of an ISP, our system overcomes two key challenges: i) discovery of routing events that affect an ISP from end systems; and ii) inference of the cause of routing events based on observations from end systems. We present the details of our approach and its limitations in terms of coverage, probing granularity, and missed routing attributes in §3.

We have designed and implemented a system that diagnoses routing events based on end system probing. Our system relies on collaborative probing from end systems to identify and classify routing events that affect an ISP. It models the routing event correlation problem as a bipartite graph and searches for plausible explanation of these events using a greedy algorithm. Our algorithm is based on the intuition that routing events occurring close together are likely explained by only a few causes, which do not create many inconsistencies. We also use probing results to study the impact of routing events on end-to-end path latency.

We instantiate our system on PlanetLab and use it to diagnose routing events for five big ISPs over a period of four months. Although each end-host has only limited visibility into the routing state of these ISPs, our system can discover many significant routing events, *e.g.*, hot-potato changes and peering session resets. Compared to existing ISP-centric method, our approach can distinguish internal and external events with up to 92.7% accuracy. Our system can also identify the causes for four out of the six disruptions reported from NANOG mailing lists [10] during that period.

We summarize our main contributions. Our work is the first to enable end systems to scalably and accurately diagnose causes for routing events associated with large ISPs without requiring access to any proprietary data such as real-time routing feeds from many routers inside an ISP. Unlike existing techniques for diagnosing routing events, our approach of using end system based probing creates a more accurate view of the performance experienced by the data-plane forwarding path. Our

work is an important first step to enable diagnosis of routing disruptions on the global Internet accounting for end-to-end performance degradations.

2 System Architecture

We present an overview of our system in this section. To diagnose routing events for any given ISP (which we call a **target ISP**), our system must learn the continuous routing state of the ISP. Based on the change in routing state, it identifies and classifies individual routing events. Because a single routing disruption often introduces many routing events, our system applies an inference algorithm to find explanations for a cluster of events occurring closely in time. It then uses the latency measurements in the probes to quantify the impact of these routing events. As shown in Figure 1, our system is composed of four components:

Collaborative probing: This component learns the routing state of a given ISP via continuous probing from multiple end systems. Given the large number of destinations on the Internet, the key challenge is to select an appropriate subset to ensure coverage and scalability.

Event identification and classification: This component identifies routing events from a large number of end-system probes. These events are then classified into several types based on the set of possible causes, *e.g.*, internal changes, peering failures, or external changes.

Event correlation and inference: This component searches for plausible explanation for routing events. Although each routing event may be triggered by many possible causes, we seek to identify a small set of causes that can explain all the events occurring close in time. We model the inference problem as a bipartite graph and solve it with a greedy algorithm.

Event impact analysis: This component extracts latency information from end-system probes. It enables us to study the impact of routing events on path latency according to the cause of events and the impacted ISPs. Note that this information is not readily available in routing feeds used in previous work on routing diagnosis.

3 Collaborative Probing

For a target ISP, we need to know its routing state to identify and diagnose its routing events. Unlike previous work that uses many routing feeds from a single ISP [6],

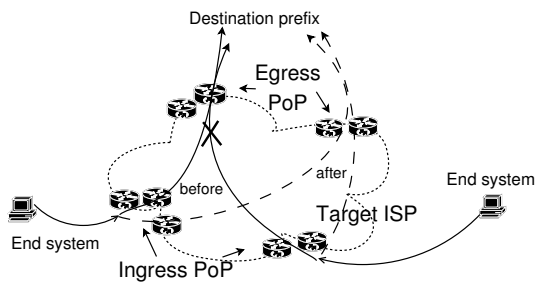


Figure 2: Collaborative probing to discover routing events.

our system relies on end systems that do not have any direct visibility into ISP's routing state. Note that it is important to obtain a comprehensive view of the routing state across major Points of Presence (PoPs) of the target ISP in order to diagnose routing events associated with the ISP. Utilizing public routing repositories is insufficient due to only one or at most two feeds from each ISP, in addition to issue of a lack of real-time data feeds. The key question in our design is how to learn the routing state of an ISP from end-system probing alone.

3.1 Learning routing state via probing

A router's routing table contains the traffic forwarding information, *e.g.*, the next hop, based on the destination prefix. Although an end system may not have direct access to the routing tables, it could learn this next hop information using *traceroute* if the forward path from the host to the destination happens to traverse the router. As illustrated in Figure 2, *traceroute* probing from two end systems to one particular destination experiences egress PoP shifts due to the target ISP's internal disruption. Ideally, we can learn the next hop from any router to any destination by probing from an appropriate source. This is not always possible because we may not have access to such a source or the router may not respond to our probes.

We focus on diagnosing inter-domain routing events that affect a target ISP. We aim to find explanations for events that the ISP should be held accountable for and can directly address, *e.g.*, internal routing changes and peering session failures. For internal or intra-domain routing events it is obvious which ISP should take responsibility for them. Therefore, we do not focus on constructing detailed intra-domain routing tables. Instead, we keep track of the inter-domain routing tables (BGP tables) of each major PoP within the ISP.

There are three challenges associated with constructing BGP tables. First, given a limited set of end systems, the system attempts to obtain as many routes between PoP-prefix pairs (PoP to destination prefix) as possible. Second, end systems have limited resources (CPU and network), and our system must have low probing over-

head. Third, probing needs to be launched frequently to accurately track the dynamic routing state.

To address the first two challenges, we devise a scheme to select an appropriate set of destinations for each end system to probe. We start with a set of prefixes extracted from BGP tables. Each end system acquires its own routing view by conducting *traceroute* to one IP in each of these prefixes. Using the existing method developed in Rocketfuel [11], we can infer whether each *traceroute* probe goes through the target ISP and the PoPs traversed. Combining the routing views from all the end systems, we obtain a complete set of PoP-prefix pairs visible to our system. We then try to select a minimum set of *traceroute* probes that can cover all the visible PoP-prefix pairs with a greedy algorithm. At each step, we select a *traceroute* probe that traverses the maximum number of uncovered PoP-prefix pairs and remove these newly-covered pairs from the set of uncovered pairs. This process continues until there is no uncovered PoP-prefix pair left. The selection process has been shown to be effective in balancing between coverage and overhead [12]. Note that because ISP network topology and routing evolve over time, each end system periodically refreshes its routing view. Currently, this is done once a day to achieve a balance between limiting probing overhead and capturing long-term changes.

To address the third challenge, we developed a customized version of *traceroute* which enhances the probing rate by measuring multiple destinations and multiple hops in parallel up to a pre-configured maximum rate. To prevent our measurement results from being affected by load-balancing routers, all probe packets have the same port numbers and type of service value. With our improvement, all the end systems can finish probing their assigned set of destinations in roughly *twenty minutes*. This also means that our system can obtain a new routing state of the target ISP every twenty minutes, the details of which are shown in §6.

3.2 Discussion

Although learning an ISP's routing state via collaborative probing does not require any ISP proprietary information, it has three major limitations compared with direct access to BGP routing feeds: (i) given a limited number of end systems, we cannot learn the route for every PoP-prefix pair; (ii) given limited CPU and network resources at end systems, we cannot probe every PoP-prefix pair as frequently as desired. This implies we may miss some routing events that occur between two consecutive probes; and (iii) we can only observe forwarding path changes but not other BGP attribute changes.

The first problem of coverage is a common hurdle for systems finding root causes of routing changes as described by Teixeira and Rexford [9]. They presented an

idealized architecture for cooperative diagnosis which requires coverage in every AS. Similar to the work by Wu *et al.*, our work addresses a simpler problem of diagnosing routing changes associated with a large ISP but purely from end system's perspectives. Our ability to address this relies on the coverage obtained.

A straightforward solution to improving coverage is to use more end systems. In this paper, we use all the available PlanetLab sites (roughly 200) to probe five target ISPs. We will explain the detailed coverage results in §6. Note that a single major routing disruption near the target ISP, *e.g.*, a hot-potato change or a peering session failure, often introduces a large number of routing events and affects many different PoPs and prefixes. In §7, we will show that our system is able to correctly identify many such major disruptions despite covering only a subset of the affected PoP-prefix pairs. As future work, we plan to study how better coverage will improve our inference accuracy. Besides the coverage limitation, topology discovery could be affected by ISPs' ICMP filtering policy. Fortunately, we find this is performed mostly by ISPs on their edge routers connecting to customers, which has little impact on our inference.

We consider the second problem of limited probing frequency to be less critical. Our system focuses on diagnosing routing changes that are long-lived enough to warrant ISP's corrective action rather than transient ones that may repair by themselves quickly. Reporting every transient event may actually overwhelm ISP operators.

The third problem is more fundamental to systems that rely on end-system probing, given that BGP data can be inherently proprietary. This implies we might identify or locate a routing change but might not know *why* it occurs. We give an example of this in §5 where we cannot distinguish a route change triggered by different attribute changes. The focus of our work is on determining whether an ISP should be held accountable for a routing problem and providing useful hints for the ISP to diagnose it. We believe the responsible ISP can subsequently use its own data to perform root cause analysis.

4 Event Identification and Classification

In this section, we first describe how we identify individual routing events from the time sequence of routing state captured for a target ISP. We then present our event classification method based on likely causes.

4.1 Data processing

As explained in the previous section, we focus on the inter-domain routing state of the target ISP. Given a PoP-prefix pair, we identify the next hop and the AS path from the PoP to the destination prefix. The next hop can be either a PoP in the target ISP or another ISP. This implies that we need to extract the ISP and PoP information

from end systems' traceroute probes.

A traceroute probe only contains the router's interface address along the forwarding path from the source to the destination. We map an IP address to a PoP in the target ISP using the existing tool based on DNS names (*undns*) [13]. For instance, 12.122.12.109 reverse-resolves to *tbr2-p012601.phlpa.ip.att.net*, indicating it is in the AT&T network, located in Philadelphia (phlpa). *undns* contains encoded rules about ISPs' naming conventions. For IP addresses not in the target ISP, we map them to ASes based on their origin ASes in the BGP tables [14]. One IP address may map to multiple origin ASes (MOAS) and we keep a set of origin ASes for such IP addresses. After performing IP-to-PoP and IP-to-AS mappings for each traceroute probe, we know the traversed PoPs in the target ISP and the AS path to the destination prefix. Given that errors in IP-to-AS and IP-to-PoP mappings are sometimes inevitable, we present a greedy algorithm that lowers the number of incorrect mappings by reducing total conflicts in event correlation and inference (§5).

Note that not all traceroute probes are used for routing event identification and classification. They may be discarded for several reasons:

Not traversing the target AS: Traceroute probes may not traverse the target ISP when the source hosts do not have up-to-date routing views or the probes are conducted during temporary routing changes. Such probes are discarded because they do not contribute any routing information about the target ISP.

Contiguous "*" hops: Traceroute paths may contain "*" hops when routers do not respond to probes due to ICMP filtering or rate-limiting. A "*" hop is treated as a wildcard and can map to any ISP or PoP. To simplify path matching for event identification, we discard traceroute containing two or more consecutive "*" hops.

Loops: Traceroute paths may contain transient loops that likely capture routing convergence. Such traceroute paths are not stable and somewhat arbitrary because they depend on the subtle timing when routers explore alternate paths. Since our goal is to infer the likely causes of routing events, we are interested in the stable paths before and after a routing event rather than the details of the transition. We discard traceroute paths that contain IP-level, PoP-level, or AS-level transient loops.

Some traceroute paths may contain loops that persist for more than 20 minutes. Since most routing convergence events last for several minutes [15], these loops are likely caused by routing misconfigurations [16] rather than unstable router state during convergence. We still make use of such traceroute paths after truncating their loops, since the partial paths represent stable paths.

4.2 Event identification and classification

We now describe how we identify inter-domain routing events that affect the target ISP from the continuous snapshots of routing state obtained from traceroute probes. An *inter-domain routing event* is defined as a path change from a PoP to a destination prefix, in which either the next hop or the AS path has changed. Since our system acquires a new routing state of the target ISP periodically, we can identify an event by observing a path change between the same source and destination in two consecutive measurements.

Given that there could be “*” hops and multiple-origin-ASes (MOAS) hops, we choose to be conservative in comparing two paths by trying to search for their best possible match. For instance, $path(A, *, C)$ is considered to match $path(A, B, C)$ because “*” can match any ISP or PoP. Similarly, a MOAS hop can match any AS in its origin AS set.

When observing path changes between two consecutive measurements, we classify them into three types according to their likely causes. The classification is motivated by our goal of inferring the causes of the changes relative to the target ISP.

Type 1: Different ingress PoP changes can be caused by routing events in the upstream ISPs, the target ISP, or downstream ISPs. Realizing it is difficult to enumerate all possible causes, we do not currently use them for event correlation and inference.

Type 2: Same ingress PoP but different egress PoP changes can be caused by internal disruptions in the target ISP (e.g., hot-potato changes), failures on its border (e.g., peering session reset), or external changes propagated to the target ISP (e.g., prefix withdrawals).

Type 3: Same ingress PoP and same egress PoP changes are easier to deal with compared to the previous two types. They may involve internal PoP path changes, external AS path changes, or both. We will explain how to use such information for event correlation and inference in the next section.

5 Event Correlation and Inference

It is well known that a single major routing disruption often leads to a burst of routing events and affects many PoPs and prefixes simultaneously. Our goal is to diagnose which inter-domain routing events are triggered by those major disruptions that the target ISP should be held accountable for and can take action on.

In many cases, it is extremely difficult to infer the cause of an individual routing event because an event may be explained by many different causes. An obvious solution is to improve inference accuracy by correlating multiple “relevant” events together. However, the key

1. Ignore if the next hop is unreachable
2. Highest local preference
3. Shortest AS path
4. Lowest origin type
5. Lowest Multiple-Exit-Discriminator (MED) value among routes from the same AS
6. eBGP learned route over iBGP learned route
7. Lowest IGP cost (hot-potato)
8. Lowest router ID

Table 1: BGP decision process

question is how we can discover and make use of the relevancy among events.

5.1 Inference model

Before describing our inference model used for event correlation, we make an assumption that each routing event can be explained by only one cause. This is a standard assumption made in many existing work on root cause analysis [5, 9] and fault diagnosis [17]. Note that this assumption does not prevent us from inferring multiple simultaneous causes as long as the events triggered by different causes are independent.

We start by defining some terminology to facilitate our discussion. Since each event is identified by observing the change between two consecutive probes, we call the earlier path probe an **old path** and the later one a **new path**. We call the egress PoP on the old/new path the old/new egress respectively. In the previous section, we classify individual routing events into three types. Currently, we do not use the events of the first type for correlation because it is infeasible to enumerate all the possible causes for them. We identify all the possible causes for the latter two types of events based on how BGP selects a single best route for each prefix. When multiple routes are available, BGP follows the decision process in Table 1 to select the best one.

Same ingress PoP but different egress PoP changes can be triggered by a prefix withdrawal, a prefix announcement, or a change in any of the eight steps in Table 1. We ignore *Step₈* since router ID rarely changes. *Step₆* is irrelevant because both the old and the new egress use external paths. The following causes comprehensively cover all the remaining possibilities:

- A change in *Step₁* is explained by either an *Old-Peering-Down* or a *New-Peering-Up*. The former implies the peering between the old egress and its neighbor AS is down. The latter means the peering between the new egress and its neighbor is up.
- A change in *Step₂* can be explained by either an *Old-Lpref-Decrease* or a *New-Lpref-Increase*. The former implies the local preference (*Lpref*) at the old egress decreases. The latter implies the *Lpref* at the new egress increases.

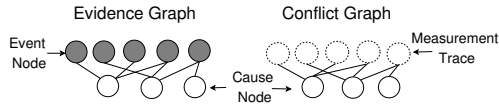


Figure 3: The bipartite graphs for cause inference

- A prefix withdrawal, an announcement, or a change in *Step₃₋₅* can be explained by either an *Old-External-Worsen* or a *New-External-Improve*. The former means the old route to the prefix worsens due to an external factor (e.g., a prefix withdrawal, a longer AS path, a higher origin type, or a higher MED value). The latter implies the new route to the prefix improves due to a prefix announcement, a shorter AS path, a lower origin type, or a lower MED value.
- A change in *Step₇* can be explained by an *Old-Internal-Increase* or a *New-Internal-Decrease*. The former implies the cost of the old internal path increases due to a more costly PoP-level link. The latter implies a less costly new internal path.

Same ingress PoP and same egress PoP changes

- When the internal PoP path changes, it can be explained by an *Old-Internal-Increase* or a *New-Internal-Decrease*.
- When the next hop AS changes, it can be explained by an *Old-Peering-Down*, a *New-Peering-Up*, an *Old-External-Worsen*, or a *New-External-Improve*.
- When the AS path changes but no next hop AS changes, it can be due to an *External-AS-Change*, which is not directly related to the target ISP.

Using the above rules, we can map each event to a set of possible causes. By aggregating events that occur closely in time (identified between the same pair of consecutive routing state), we construct a *bipartite graph*, called *evidence graph*, as shown in Figure 3. There are two types of nodes in an evidence graph: cause nodes at the bottom and event nodes at the top. An edge between a cause node and an event node indicates the event can be explained by the cause. An evidence graph encapsulates the relationship between all the possible causes and their supporting evidence (events).

Conflicts may exist between causes and measurement traces due to noise and errors. For instance, an *Old-Peering-Down* will conflict with a new trace which traverses the peering that is inferred to be down. Conflicts stem from two major sources: i) the subtle timing difference when traceroute probes from different end systems traverse the same peering or measure the same prefix; and ii) errors in the IP-to-AS or IP-to-PoP mappings.

A measurement trace will never conflict with an *Old-Internal-Increase* or a *New-Internal-Decrease* because a cost change on a PoP-level link may not prevent a path from using the link. However, a measurement trace may

conflict with each of the remaining six causes:

- *Old-Peering-Down*: a new path still uses a peering that is inferred to be down.
- *New-Peering-Up*: an old path already used a peering that is inferred to be up.
- *Old-Lpref-Decrease*: a new path still uses an egress that is inferred to have a lower *Lpref* even when there are other egresses with a higher *Lpref*.
- *New-Lpref-Increase*: an old path already used an egress that is inferred to have a higher *Lpref* (therefore used to have a lower *Lpref*) even when there were other egresses with a higher *Lpref*.
- *Old-External-Worsen*: a new path still uses an old route to a prefix even when it is worse than a new route to the same prefix, or an old path already used a new route to a prefix even when the old route to the same prefix was better.
- *New-External-Improve*: a new path still uses an old route to a prefix even when a new route to the same prefix is better, or an old path already used a new route to a prefix even when it was worse than an old route to the same prefix.

We encapsulate the relationship among all the possible causes and their conflicting measurement traces using a *conflict graph*, as shown in Figure 3. Similar to an evidence graph, it has two types of nodes: cause nodes at the bottom and measurement nodes at the top. An edge between a cause node and a measurement node indicates a conflict between the cause and the measurement trace. For each evidence graph, we construct a conflict graph accordingly by inspecting all the measurement traces in the same pair of consecutive routing state. When a measurement trace conflicts with some causes in the evidence graph, we insert a measurement node and the corresponding edges into the conflict graph.

5.2 Inference algorithm

We now present our inference algorithm that uses the evidence graph and the conflict graph to infer likely causes. Our inference is guided by two rules: i) Simplest explanation is most likely to be true. We try to find the minimum set of causes that can explain all the evidence (events). ii) We should take into account the noise and errors in our measurement by minimizing conflicts between inferred causes and measurement traces.

We use a greedy algorithm to infer causes. In each iteration, it selects a cause from the evidence graph with the maximum value of $(E - \alpha C)$, where E is the number of supporting events and C is the number of conflicting traces (computed from the conflict graph). Intuitively, it selects a cause that explains many events but raises few conflicts. It then removes the events that have been explained by the cause from the evidence graph before entering the next iteration. This process continues until

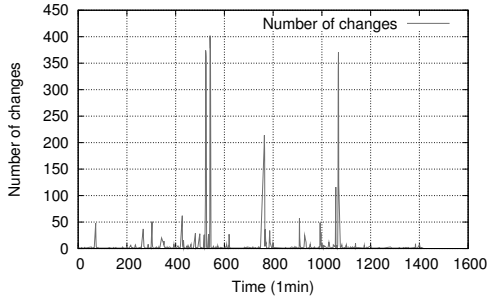


Figure 4: Number of detected changes on Sep. 25, 2007

AS Name ASN (Tier)	Periods	# of Src	# of PoPs	# of Probes	Probe Gap
AT&T 7018 (1)	3/23-4/9	230	111	61453	18.3 min
Verio 2914 (1)	4/10-4/22 9/13-9/22	218	46	81024	19.3 min
Deutsche Tele -kom 3320 (2)	4/23-5/22	149	64	27958	17.5 min
Savvis 3561 (1)	5/23-6/24	178	39	40989	17.4 min
Abilene 11537 (3)	9/23-9/30 2/3-2/17	113	11	51037	18.4 min

Table 2: Summary of data collection

all the events have been explained.

The parameter α allows us to tune the relative weight between evidence and conflicts. A larger α makes our algorithm more aggressive in avoiding conflicts. Currently, we set $\alpha = 1$ in our experiments. However, we find our results are not very sensitive to the choice of α between 0.1 and 10. This is likely due to the fact that the number of evidence significantly outweighs the number of conflicts for most causes (see §7).

Given that the inputs to our algorithm (the evidence graph and the conflict graph) are limited by the coverage of our system and measurement noise and errors, it may report incorrect causes or miss true causes. To highlight the reliability of inferred causes, we introduce a notion of *inference confidence* for each cause as $E - \alpha C$, where E and C have the same meaning as in the above. Intuitively, causes with a higher inference confidence, *i.e.*, with more evidence but fewer conflicts, are more reliable. We will demonstrate how inference confidence affects the accuracy in §7.

6 Results of Event Identification and Classification

In this section, we present the results of event identification and classification using our framework over a period of 132 days for five backbone ISPs. We validate the identified routing events using BGP data from many vantage points at the end of the section.

The summary of data collection is shown in Table 2. We study three Tier-1 ASes, one Tier-2 AS, and one Tier-3 AS. As the first step, we study one AS at a time.

We plan to study multiple ASes simultaneously in the future to better diagnose routing events at a global scale. Table 2 shows the number of probing source hosts used and the number of PoPs covered. Note that there is some variability across the number of source hosts used as not all hosts are useful for improving the coverage of PoP-prefix pairs. This provides room for probing multiple ASes at the same time. We verified our PoP coverage completeness using the data from Rocketfuel [11] and router configuration files from the Abilene network. Table 2 also shows the average number of probes to acquire the routing state of a target ISP. Depending on the ISP, each source host has to probe between 187 and 371 destinations on average. As expected, our system can refresh the routing state roughly every *eighteen* minutes.

Before delving into details, we first use one example to illustrate that our system is able to detect significant network disruptions that generate a large number of routing events. Figure 4 shows the number of routing events detected using our system for Abilene over time on Sep. 25, 2007. It is clear that the routing event occurrence is not evenly distributed. We do observe a few spikes across the day. The constant background noise is often due to routing events that only affect individual prefixes. The spike around 540min is an internal disruption causing the egress PoP to shift from Washington DC to New York, affecting 782 source-destination pairs. The next spike around 765min is due to one neighbor AS2637 withdrawing routes to 112 prefixes from the Atlanta PoP. The last spike around 1069min is due to a peering link failure, resulting in the next hop AS in Washington DC changing from AS1299 to AS20965. All these causes have been confirmed using the BGP and the Syslog data of Abilene.

6.1 Data cleaning process

As mentioned in §4, we first need to remove the noise in our dataset. Table 3 shows the overall statistics of average daily traces removed due to various reasons. It is expected that a relatively small percentage (0.75%) of traces are ignored due to contiguous “*” hops and temporary loops. We also found that 0.025% of the traces contain persistent IP or AS loops usually occurring close to the destination, which confirms observations from a previous study [16].

Note that 3.2% of the traces are discarded due to not traversing the target ISP, as we cannot distinguish between the target ISP losing reachability or any of the preceding ISPs changing routes. One noteworthy observation is that 35% of the traces stop before entering the destination network. Most of these networks appear persistently unreachable over time, likely due to ICMP filtering at the edges between a provider and its customers. We still use these traces as they can help detect routing

	IP loop	PoP loop	AS loop	IP star	PoP star	AS star	No targetAS	Persistent IP loop	Persistent AS loop
Removed traces (percentage)	12643 0.18%	9934 0.14%	1053 0.015%	14055 0.2%	5836 0.08%	9573 0.13%	2466927 3.2%	1738 0.02%	445 0.005%

Table 3: Statistics of data cleaning: avg number of removed traces per day for each type of anomalous traceroute.

Target AS	Total events (% all traces)	Ingress same Egress change	Ingress same, Egress same		Ingress change
			internal pop path	external AS path	
7018	277435 0.35%	33325 12.1%	213562 51%	76.9% 35%	30548 11%
2914	415778 0.31%	113507 27.3%	261525 48%	62.9% 19%	40746 9.8%
3320	437125 0.66%	21419 4.9%	384233 8.5%	87.9% 80.7%	31473 7.2%
3561	311886 0.35%	34307 11%	233915 45%	75% 31%	43664 14%
11537	145034 0.24%	19776 13.6%	99309 37%	68% 40%	25949 17%

Table 4: Statistics of event classification

changes in the partial path before filtering.

6.2 Event identification and classification

We first classify routing events according to the ingress and egress PoP changes. Table 4 shows the statistics of event classification for each ISP during our study. Only a very small fraction of the traces contain routing changes. Among these changes, a small percentage (7.2% - 17%) is found to be ingress PoP changes, because most of the probing sources enter the target AS from an ingress PoP near its geographic location. The majority (62.9% - 87.9%) of the events are in the category of both ingress and egress staying the same. This category contains either internal PoP-level path changes and/or the external AS path changes. The remaining events (4.9% - 27.3%) involve egress PoP changes. Some of these events may impose significant impact on the target ISP as a large amount of traffic to many prefixes shifts internal paths simultaneously.

Abilene, the educational backbone network, was expected to be stable due to its simple topology. Surprisingly, we found that it has a larger fraction of ingress changes. This is observed mainly from three source hosts, switching their ingress PoP to various destinations. Two of them are universities in Oregon, with access links to Abilene in both Seattle and Los Angeles. The other one is a university in Florida, which has access links in both Atlanta and Kansas City. We confirm this via the Abilene border routers' configuration files. We believe this could be due to load-balancing or traffic engineering near the sources.

6.3 Validation with BGP data

Using public BGP feeds from RouteViews, RIPE and Abilene, in addition to 29 BGP feeds from a Tier-1 ISP,

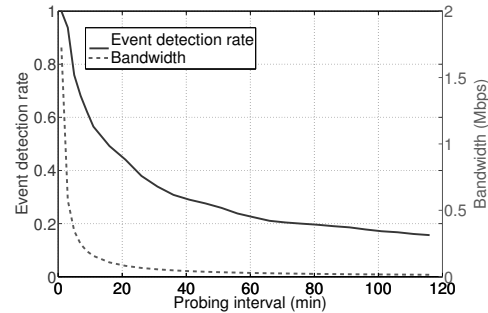


Figure 5: Impact of probing interval on detection rate and bandwidth.

we validate our results in two aspects: the *destination prefix coverage* and the *routing event detection rate*. We omit AS3320 here due to lack of access to its BGP data.

To evaluate the destination prefix coverage of our dataset, we map the destination IP to the longest prefix using the latest routing table of each AS. Then by comparing the set of probed prefixes with all the prefixes in the default-free routing table of each target AS, we compute the coverage, as shown in the second column of Table 5. Although the coverage is only between 6% to 18%, our traces cover all the known distinct PoP-level links within each target AS (compared to the Rocketfuel data [11]), suggesting that we can detect significant routing changes originated inside the target AS.

We use the following methodology for validating changes detected using BGP data. For the five ASes we studied, we only have BGP feeds for four ASes. For each of them, we first identify the corresponding PoP where the BGP feed comes from. Because different PoPs in an AS usually experience different routing changes, we compare BGP-observed changes with traceroute-observed changes only when our traces traverse the PoP where the BGP feed comes from. The third column of Table 5 shows the ratio of the probed destination prefixes that traverse the PoP of the BGP feed relative to the total number of prefixes in a default-free routing table.

The subset of destinations which can be used for comparison varies across ASes due to the different number of available BGP feeds. We focus on examining for any BGP-observed routing change of this subset of destinations, whether we also detect it using our traces. Moreover, we only account for BGP routing changes with either AS path changes or next hop AS changes, which can

Target AS	Dst. prefix coverage	Dst. prefix traversing PoPs with BGP feeds	Detected events (AS change, nexthop change)	Missed events (short duration, filtering, others)
7018	34145 (15%)	3414 (1.5%)	64714, 11% (10.3%, 3.2%)	89% (75%, 13%, 1%)
2914	40881 (18.6%)	40039 (18.1%)	73689, 23% (19.1%, 8.6%)	77% (73%, 4%, 0%)
3561	17317 (7.8%)	2317 (1.1%)	55692, 6% (5.8%, 0.5%)	94% (80%, 9%, 5%)
11537	13789 (6%)	13789 (6%)	66706, 21% (17.3%, 5.8%)	79% (61%, 15%, 3%)

Table 5: Validation with BGP data for dst. prefix coverage and event detection rate.

be detected via traceroute. By comparing the two sets, we calculate the fraction of changes our system can detect, as shown in the fourth column of Table 5. This rate varies between 6% to 23%. Note that we can also detect many internal PoP path changes which are not observed in the BGP data (thus not included in this table).

Changes missed by our system are due to two main reasons. First, the routing changes last too short to be detected by two consecutive probes, accounting for the majority of the missed routing events. As explained in §3, we do not focus on these short-lived routing events. We are able to detect most events with duration larger than 20 minutes (probing interval). Given that we cannot detect routing events that last shorter than the probing interval, we may increase the event detection rate by reducing the probing interval. Figure 5 illustrates how the probing interval affects the event detection rate and probing bandwidth. When the probing interval is 10 minutes, we can detect 60% of the events while using roughly 0.2 Mbps bandwidth.

Second, because traceroute may be incomplete due to packet filtering, certain changes cannot be detected as the changing path segment is invisible from our probes. Most filtering happens in the path segment after the next hop AS and close to the destination AS. Since we only use the next hop AS information for event correlation, missing these changes does not have any impact on our inference results.

Only a small fraction (up to 5%) of the missed changes are due to other factors, *e.g.*, inaccurate IP-to-AS mappings or mismatched forward paths compared to the BGP data. In summary, our system is able to capture most routing changes to the probed destinations that are useful for event correlation and inference.

7 Results of Event Correlation and Inference

In this section, we first present the results of our inference algorithm. Then we validate our system in three ways: comparing with the BGP feed based inference using BGP data from a Tier-1 ISP, comparing with both BGP data and Syslog data from the Abilene network, and comparing with disruptions reported from the NANOG email list [10].

7.1 Result summary

Our inference algorithm takes the set of identified events and automatically clusters them based on their causes. Table 6 shows both the total number and the relative percentage for each type of causes inferred for each ISP. We observe that different ISPs can have a non-negligible difference in the cause distribution. For example, for the first three ISPs, the largest fraction of events are caused by *External-AS-Change*. In contrast, Abilene (AS11537) has more events caused by *Old-External-Worsen* and *New-External-Improve*. This is mainly caused by its five neighbor ASes. The most dominant one is the neighbor AS20965 peering in New York which switches routes to around 390 destinations frequently over time.

We study the effectiveness of our inference algorithm in clustering related events together in Figure 6(a). A cluster is defined to be the set of events explained by a single cause. The figure shows the CDF of the number of events per cluster over the entire period for five ASes. While most of them have less than ten events per cluster, there are some clusters with many events, indicating significant routing disruptions. *New-Internal-Decrease*, *Old-Internal-Increase*, *Old-Peering-Down*, and *New-Peering-Up* have relatively larger clusters than others, confirming previous findings that hot-potato changes and peering session up/down can impose significant impact [18]. Other types of causes have much smaller clusters, because they usually only affect individual prefixes.

Another metric to evaluate the accuracy of inferred cause is based on the number of conflicts introduced by the cause, as shown in Figure 6(b). According to §5, only six types of causes may have conflicts. Overall, the number of conflicts per cluster is small compared to the number of events per cluster, indicating that the inconsistencies in our traces introduced by incorrect mappings or differences in probing time are rare.

We use the confidence metric introduced in the previous section to assess the likelihood of causes. Figure 6(c) shows that different types of causes have different distributions of confidence value. For example, *Old-External-Worsen*, *New-External-Improve*, *Old-Lpref-Decrease*, and *New-Lpref-Increase* generally have much lower confidence values as they affect only indi-

Target AS	Old-Int. -Increase	New-Int. -Decrease	Old Peer-ing Down	New Peer-ing Up	Old-Ext. -Worsen	New-Ext. -Improve	Old-Lpref -Decrease	New-Lpref -Increase	Ext. AS Change
7018	5223, 4.5%	3843, 3%	5677, 5%	4955, 4.3%	18142, 16%	20961, 18%	302, 0.2%	397, 0.3%	55216, 48%
2914	10366, 5%	8135, 4%	6666, 4%	7024, 3.7%	38748, 20%	49075, 26%	124, 0.1%	164, 0.1%	69190, 36%
3320	1622, 0.5%	954, 0.2%	20751, 5%	10204, 3%	80385, 21%	81761, 21%	751, 0.2%	1002, 0.2%	185683, 48%
3561	4410, 3.6%	4007, 3%	6017, 5%	7667, 6.3%	23232, 19%	45495, 37%	85, 0.1%	105, 0.1%	30540, 25%
11537	2161, 1.8%	1632, 1%	2771, 2%	1401, 1.1%	44516, 37%	43375, 36%	112, 0.1%	104, 0.1%	9589, 8%

Table 6: Statistics of cause inference.

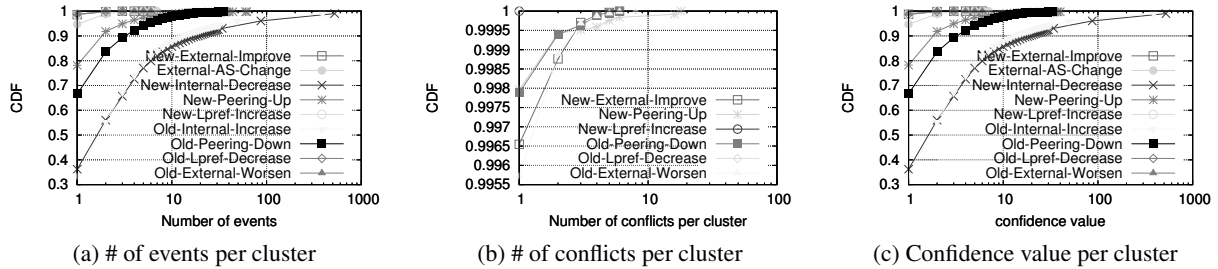


Figure 6: Events, conflicts and confidence value distribution per cluster.

vidual prefixes. Thus we need to set appropriate thresholds to filter out different types of causes with low confidence. Throughout the rest of this section, we use a confidence value of 30 for reporting hot-potato changes (*Old-Internal-Increase* and *New-Internal-Decrease*) and 150 for reporting peering session changes (*Old-Peering-Down* and *New-Peering-Up*). A lower confidence value increases the likelihood of false positives, *e.g.*, misinterpreting multiple simultaneous prefix withdrawals from a peering as an *Old-Peering-Down*. These two confidence values filter out 92% of the hot-potato changes and 99% of the peering session changes inferred without using any thresholds. Next we evaluate the impact of the confidence value on our inference accuracy. We do not set any threshold for other types of causes since most of them have only one event in each cluster.

7.2 Validation with BGP-based inference for a Tier-1 ISP

Most previous work on diagnosing routing disruptions relies on BGP data. The closest one to ours is by Wu *et al.* [6] using BGP updates from all the border routers to peers to identify important routing disruptions. To directly compare with their approach, we implemented their algorithm, called *Wu* for convenience. We collected data via eBGP sessions to 29 border routers in a Tier-1 ISP. Note that *Wu* requires BGP data from all the border routers and focuses on peer routes only. Given the lack of access to such complete data, causes reported by *Wu* on our data may be inaccurate accounting for possible mismatches.

We briefly summarize *Wu*'s algorithm and our comparison methodology. *Wu* first groups a routing event from one border router's perspective into five types: no change, internal path change (using iBGP routes with

next hop change), loss of egress point (changing from eBGP to iBGP route), gain of egress point (changing from iBGP to eBGP route), and external path change (both using eBGP route with next hop change). This step is accurate even with incomplete data. By correlating events from individual routers, *Wu* generates a vector of events for each destination prefix to summarize how the route for each prefix has changed. The types of changes include: *transient disruption*, *internal disruption* (all routers experience internal path change), *single external disruption* (only one router has either loss/gain of egress or external change), *multiple external disruption* (multiple routers have either loss/gain of egress or external changes), and *loss/gain of reachability* (every router experiences loss/gain of egress). This step may introduce inaccuracy due to data incompleteness. Note that incomplete data set can only cause *Wu* to falsely categorize external events into internal events.

We first validate our event classification results by comparing with *Wu*'s vector change report. We map each of our events (per source-destination based routing change) to the corresponding event in *Wu*, the prefix of which covers our destination. Each event is associated with one cause from our algorithm and one vector change type in *Wu*. Note that the set of causes and the set of vector change types do not have direct one-to-one mapping. To perform comparison, we combine our causes into two big categories:

Internal includes *New-Internal-Decrease*, *Old-Internal-Increase*, *Old-Lpref-Decrease*, *New-Lpref-Increase*, which should match *Wu*'s *internal disruption*.

External includes *Old-External-Worsen*, *New-External-Improve*, *Old-Peering-Down*, *New-Peering-Up*, which should match *Wu*'s *single/multiple external*

Root cause	Internal disruption	Single external	Multiple external	Loss/gain of reachability
Internal	34914 (76.9%)	5947 (13.1%)	4494 (9.9%)	10 (0.02%)
External	16344 (24.2%)	44948 (65.9%)	6538 (9.6%)	391 (0.6%)

Table 7: Event based validation: with a Tier-1 ISP’s BGP data over 21 days.

disruption.

These two aggregate categories are of interest because our main goal is to distinguish internal disruptions from external ones. The cause *External-AS-Change* does not have any corresponding type in *Wu*, which is thus omitted from comparison. Similarly, we omit our Same-Ingress-Same-Egress type of events with only internal PoP path changes, as it is not considered by *Wu*.

As shown in Table 7, each column is the type of vector change in *Wu*, while each row shows our aggregate categories. For each routing event, we identify the type y inferred from *Wu* as well as the category x inferred by our system. By comparing them, we generate the percentage in the table row x column y which is the fraction of events in our aggregate category x that is categorized as type y in *Wu*. The cell with bold italic font means valid matches. 76.9% of our internal events match *Wu*’s *internal disruption*, while 75.5% of our external events match *Wu*’s *single/multiple external disruption*. While the match rate of around 75% is not very high, we believe our end-system based approach shows promise in inferring routing disruptions and the rate can be further improved with more vantage points.

The third step in *Wu* is to group together event vectors of different destinations belonging to the same type and transition trend. There are two types of clusters reported in the third step: hot-potato changes and peering session resets. For each of the causes reported by us, we examine if it is also reported by *Wu*. To be more specific, for each *New-Internal-Decrease* and *Old-Internal-Increase*, we search for the corresponding hot-potato changes reported within that probing interval. Each *Old-Peering-Down* and *New-Peering-Up* is mapped to *Wu*’s *peering session reset* in the same probing interval associated with the same egress and neighbor AS.

The comparison for these two important clusters is shown in Table 8. We use the confidence value of 30 for hot-potato changes and 150 for session resets based on their distinct confidence distributions shown in the previous section. The two algorithms reported 101 common hot-potato changes and 6 common session resets. Given that our system does not rely on any ISP proprietary data, it is quite encouraging that we can correctly diagnose a reasonably large fraction of significant rout-

Target AS	Hot potato			Session reset		
	<i>Wu</i>	Our	Both	<i>Wu</i>	Our	Both
Tier-1 ISP	147	185	101 68%,55%	9	15	6 66%,40%
Abilene (11537)	79	88	60 76%,68%	7	11	7 100%,63%

Table 8: Validation for two important clusters ($confidence_{hotPotato}=30$, $confidence_{session}=150$)

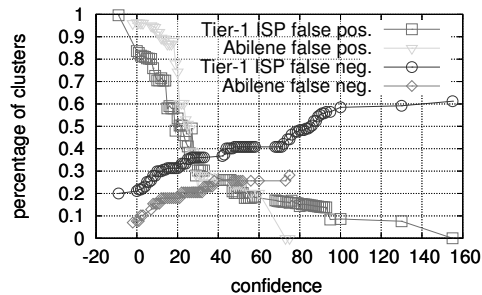


Figure 7: Inference accuracy for hot-potato changes – a common type of routing disruption.

ing disruptions (68% of hot-potato changes and 66% of session resets).

We study the impact of confidence value on our inference accuracy of hot-potato changes in Figure 7. As expected, with larger confidence values, the false positive rate decreases while the false negative rate increases. With a confidence threshold of 30, we attain a balance between false positives (45%) and false negatives (32%). Similarly, for session reset, the false positive and false negative rates are 60% and 34% respectively with a confidence value threshold of 150.

7.3 Validation with BGP-based inference and Syslog analysis for Abilene

We also validate our inference results with *Wu*’s algorithm executed on the BGP data from all 11 border routers of the Abilene network [19]. This provides a more complete view of routing changes for the entire network compared to the Tier-1 ISP case. Besides BGP data, router Syslog messages are also available [19] from all the Abilene border routers. Syslog reports error messages such as link down events due to hardware failure or maintenance. We can thus validate inferred link up/down causes directly using Syslog messages.

Table 9 compares the routing event inference between *Wu* and our system. The match rate for Abilene is higher compared to the Tier-1 ISP case, due to the improved accuracy of *Wu* given full visibility. 7.3% of the internal disruptions are mis-classified as external disruptions, most likely due to the limited coverage of our system. When an internal path is traversed only a few times, it

Cause	Internal disruption	Single external	Multiple external	Loss/gain of reachability
Internal	4463 (85%)	1059 (7.2%)	837 (8%)	2% (0.01%)
External	2929 (7.3%)	21642 (86.4%)	2355 (6.2%)	79 (0.1%)

Table 9: Event based validation: with Abilene’s BGP data over 21 days.

is less likely to be selected by our greedy algorithm as the cause of routing events. This problem could be mitigated by using more vantage points or increasing the confidence level threshold.

The comparison for the two important clusters is shown in Table 8. From the Abilene Syslog, the seven session resets were caused by peering link down events which lasted for more than fifteen minutes, possibly due to maintenance. Overall, we correctly inferred 76% of the hot-potato changes and 100% of the session resets. The false positive rates are 32% for hot-potato changes and 37% for session resets respectively.

7.4 Validation with NANOG mailing list

Given that operators today often use the NANOG (North American Network Operators Group) mailing list [10] to troubleshoot network problems, we study the archives of the mailing list messages over the time period of our study. All together we analyzed 2,694 emails using keyword searches and identified six significant routing disruptions with details described below. One interesting observation is that even though we did not directly probe the problematic ASes described in the emails, we are still able to identify the impact and infer the causes relative to the target ASes for the following four events due to their wide-spread impact:

1. Apr. 25, 2007, between 19:40 to 21:20 EDT, NANOG reported a Tier-1 ISP Cogent (AS174) experienced serious problem on its peering links causing many route withdrawals. The target AS during this time was AS3320. Our system observed increased number of routing events: 120 detected events were clustered into 96 causes of *External-AS-Change*, affecting 7 sources and 118 destinations. 87 of the events were associated with 42 destinations which were Cogent’s customers. They all switched from routes traversing Cogent. Significant delay increase was also observed.

2. May 21, 2007, around 21:50 EDT, NANOG reported a backbone link fiber cut between Portland and Seattle in the Level3 network (AS3356), resulting in reachability problems from Level3’s customers. The target AS at that time was also AS3320. Our system detected 45 events clustered into 36 causes of *Old-External-Worsen*, affecting 5 probing sources and 12 destinations. They all switched from routes traversing

Level3 to those traversing AS3491 in the Seattle PoP.

3. Jun. 14, 2007, NANOG reported a core router outage around 6am EDT in the Qwest network (AS209), affecting the performance of several networks and their customers. The target AS studied at the time was AS3561. Our system reported 24 events clustered into 23 causes of *External-AS-Change* switching from paths through AS209 to those traversing AT&T (AS7018) around the outage time, affecting 6 probing sources and 24 destinations.

4. Sep. 19, 2007, 13:00 EDT, NANOG reported that 25 routers in the Broadwing network (AS6395) had a misconfiguration resulting in BGP session removal. It caused multiple single-homed customers disconnected from the Internet. Immediately after that, our system detected 81 events clustered into 64 causes of *Old-External-Worsen*, for 76 destinations from 10 sources. The target AS, AS2914, switched from the old routes traversing Level3 (AS3356) and Broadwing to new routes traversing other peers, e.g., AS209 and AS7018.

We missed two NANOG-reported events related to routing and performance disruptions during our study. The first was on May 16, 2007, from 13:10 to 14:20 EDT, related to a hardware problem on the peering link between AT&T and Broadwing in Dallas. Our system did not capture any routing changes during this time period at that location. The second event was on May 30, 2007, around 13:00 EDT, related to significant performance degradation, along with temporary loss of reachability from Sprint in the Pittsburgh area, as confirmed from Sprint. The target AS probed was AS3561. Although our system did not report routing changes related to Sprint, it did observe abnormal incomplete traces from PlanetLab hosts in Pittsburgh.

To summarize, our system may miss some localized disruptions due to limited coverage. However, it is able to capture disruptions with global impact even when they are not directly caused by the target AS being probed.

8 Performance Impact Analysis

Routing events are known to introduce disruption to network path performance. Unlike the past work that relies on routing feeds to diagnose routing events, end-host probing used in our system enables us to understand the impact of routing events on path performance. In this section, we study to what extent end-to-end latency is affected by different types of routing events and its variation cross different ISPs.

Figure 8 illustrates the latency change for different type routing events in AS7018. For clarity, we only show five types of events: *Internal (Old-Internal-Increase, New-Internal-Decrease)*, *Peering (Old-Peering-Down,*

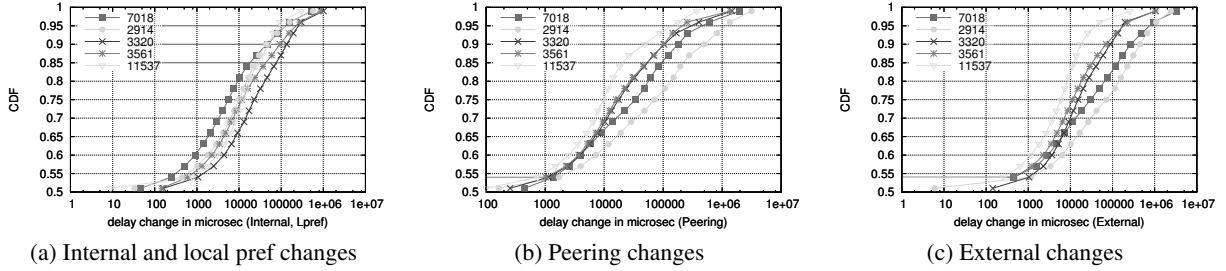


Figure 9: Delay change distribution across ISPs

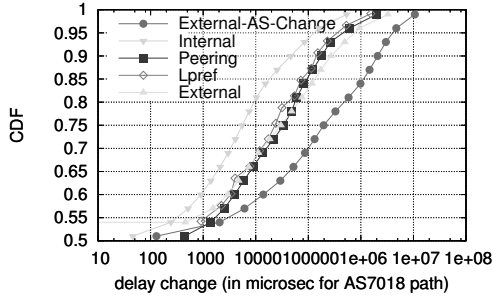


Figure 8: Delay change distribution of each category for AS7018.

New-Peering-Up), *Lpref* (*Old-Lpref-Decrease*, *New-Lpref-Increase*), *External* (*Old-External-Worsen*, *New-External-Improve*), and *External-AS-Change*. Because we use log scale on the y-axis, the graph does not show the cases where latency change is negative. Given that almost all the curves start from 0.5, it implies latency has the same likelihood to improve or worsen after these events. A noteworthy observation is external events (*External-AS-Change*, *External*, and *Peering*) have much more severe impact, suggesting that AT&T's network is engineered well internally. We observe similar patterns for the other ISPs studied.

Figure 9 illustrates how the latency change induced by the same event type varies across different ISPs. We omit *External-AS-Change* here because this type is not directly related to a target ISP. Figure 9(a) shows little difference among the five target ISPs in terms of latency change caused by internal events, as most changes are relatively small. Turning to Figure 9(b) and (c), the difference between the ISPs becomes much more noticeable. AS11537 appears most resilient to external events in terms of latency deterioration while AS2914 appears worst. The relative difference between the ISPs is consistent in both graphs, suggesting that customers sensitive to performance disruptions should take great care in selecting the appropriate ISP providers.

9 System Evaluation

In this section, we show that our system imposes a small amount of memory and CPU overhead to perform

event identification, classification, and inference. We evaluate our system on a commodity server with eight 3.2GHz Xeon processors and 4 GB memory running Linux 2.6.20 SMP.

The memory usage of our system is composed of: i) the two most recent routing state of the target ISP extracted from the traces; and ii) the evidence and the conflict graphs constructed from the two routing state (see §3). The former is relatively static over time since the overall topology and routing of a target ISP do not change frequently. The latter is more dynamic and depends on the number of detected routing events. Throughout our evaluation period, the former is dominant because the number of traces outweighs the number of routing events. The total memory footprint of our system stays under 40 MB. We also evaluate whether our system can keep up with the continually incoming routing state. We find the processing time of two recent routing state never exceeds one eighth of the data collection time between the two routing state. This suggests our system can operate in real time to quickly detect and raise alerts on significant routing disruptions.

10 Related Work

Much work has been proposed to use end-host based probing to identify various network properties. For example, Rocketfuel [11] discovers ISP topologies by launching traceroute from a set of hosts in an intelligent manner to ensure scalability and coverage. iPlane [20] estimates the Internet path performance using traceroutes and prediction techniques. There exist many other research measurement infrastructures [21, 22, 23, 24, 25] for measuring network distance with performance metrics such as latency and bandwidth. Another example is PlanetSeer [26] which uses active probes to identify performance anomalies for distributed applications. The key difference from these measurement efforts is that our work focuses on using collaborative traceroute probes to diagnose routing changes associated with large networks.

The closest related work on identifying routing disruptions is that by Wu *et al.* [6]. Using BGP data from multiple border routers in a single ISP, their system iden-

tifies significant BGP routing changes impacting large amount of traffic. A follow-up work by Huang *et al.* [27] performs multivariate analysis using BGP data from all routers within a large network combined with router configurations to diagnose network disruptions. In contrast, we do not rely on such proprietary BGP data, and we can apply our system to diagnose routing changes for multiple networks. Another closely related work is the Hubble system [28] which attempts to identify reachability problems using end-system based probing. In contrast to their work, we attempt to both identify routing events and infer their causes relative to the target AS. There are also several projects on identifying the location and causes of routing changes by analyzing BGP data from multiple ASes [5, 9]. However, it is difficult to have complete visibility due to a limited number of BGP monitors. Note that our system is not restricted by the deployment of route monitors and can thus be widely deployed.

11 Conclusion

In this paper we have presented the first system to accurately and scalably diagnose routing disruptions purely from end systems without access to any sensitive data such as BGP feeds or router configurations from ISP networks. Using a simple greedy algorithm on two bipartite graphs representing observed routing events, possible causes, and the constraints between them, our system effectively infers the most likely causes for routing events detected through light-weight traceroute probes. We comprehensively validate the accuracy of our results by comparing with an existing ISP-centric method, publicly-available router configurations, and network operators' mailing list. We believe our work is an important step to empowering customers and ISPs for attaining better accountability on today's Internet.

References

- [1] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush, "A Measurement Study on the Impact of Routing Events on End-to-End Internet Path Performance," in *Proc. ACM SIGCOMM*, 2006.
- [2] Y. Zhang, Z. M. Mao, and J. Wang, "A Framework for Measuring and Predicting the Impact of Routing Changes," in *Proc. IEEE INFOCOM*, 2007.
- [3] H. Pucha, Y. Zhang, Z. M. Mao, and Y. C. Hu, "Understanding network delay changes caused by routing events," in *Proc. ACM SIGMETRICS*, 2007.
- [4] N. Feamster, D. Andersen, H. Balakrishnan, and M. F. Kaashoek, "Measuring the effects of internet path faults on reactive routing," in *Proc. ACM SIGMETRICS*, 2003.
- [5] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs, "Locating Internet Routing Instabilities," in *Proceedings of ACM SIGCOMM*, 2004.
- [6] J. Wu, Z. M. Mao, J. Rexford, and J. Wang, "Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network," in *Proc. Symposium on Networked Systems Design and Implementation*, 2005.
- [7] "University of Oregon Route Views Archive Project." www.routeviews.org.
- [8] "Ripe NCC." <http://www.ripe.net/ripenncc/pub-services/np/ris/>.
- [9] R. Teixeira and J. Rexford, "A measurement framework for pinpointing routing changes," in *NetT '04: Proceedings of the ACM SIGCOMM workshop on Network troubleshooting*, 2004.
- [10] "NANOG Mailing List Information." <http://www.nanog.org/maillinglist.html>.
- [11] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring isp topologies with rocketfuel," *IEEE/ACM Trans. Netw.*, vol. 12, no. 1, pp. 2–16, 2004.
- [12] R. Mahajan, M. Zhang, L. Poole, and V. Pai, "Uncovering Performance Differences in Backbone ISPs with Netdiff," in *Proceeding of NSDI*, 2008.
- [13] N. Spring, D. Wetherall, and T. Anderson, "Scriptroute: A Public Internet Measurement Facility," in *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.
- [14] Z. M. Mao, J. Rexford, J. Wang, and R. Katz, "Towards an Accurate AS-Level Traceroute Tool," in *Proc. ACM SIGCOMM*, 2003.
- [15] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," in *Proc. ACM SIGCOMM*, 2000.
- [16] J. Xia, L. Gao, and T. Fei, "Flooding Attacks by Exploiting Persistent Forwarding Loops," in *Proc. ACM SIGCOMM IMC*, 2005.
- [17] R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "IP fault localization via risk modeling," in *Proc. Symposium on Networked Systems Design and Implementation*, 2005.
- [18] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford, "Dynamics of hot-potato routing in ip networks," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 307–319, 2004.
- [19] "Internet2 Network NOC - Research Data." <http://www.abilene.iu.edu/i2network/research-data.html>.
- [20] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iPlane: An Information Plane for Distributed Services," in *Proc. Symposium on Operating Systems Design and Implementation*, 2006.
- [21] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. Gryniewicz, and Y. Jin, "An Architecture for a Global Internet Host Distance Estimation Service," in *Proceedings of IEEE INFOCOM*, 1999.
- [22] T. S. E. Ng and H. Zhang, "Predicting Internet Network Distance with Coordinates-Based Approaches," in *Proceedings of IEEE INFOCOM*, June 2002.
- [23] R. Govindan and H. Tangmunarunkit, "Heuristics for Internet Map Discovery," in *Proc. of IEEE INFOCOM*, (Tel Aviv, Israel), pp. 1371–1380, March 2000.
- [24] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A Decentralized Network Coordinate System," in *Proceedings of ACM SIGCOMM*, August 2004.
- [25] M. Costa, M. Castro, A. Rowstron, and P. Key, "PIC: Practical Internet Coordinates for Distance Estimation," in *Proceedings of IEEE ICDCS*, March 2004.
- [26] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang, "PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services," in *Proc. Symposium on Operating Systems Design and Implementation*, 2004.
- [27] Y. Huang, N. Feamster, A. Lakhina, and J. J. Xu, "Diagnosing network disruptions with network-wide analysis," in *Proc. ACM SIGMETRICS*, pp. 61–72, 2007.
- [28] E. Katz-Bassett, H. V. Madhyastha, J. P. John, and A. Krishnamurthy, "Studying Blackholes in the Internet with Hubble," in *Proceeding of NSDI*, 2008.

CSAMP: A System for Network-Wide Flow Monitoring

Vyas Sekar^{*}, Michael K. Reiter[†], Walter Willinger[‡], Hui Zhang^{*},
Ramana Rao Kompella[§], David G. Andersen^{*}

^{*}Carnegie Mellon University, [†]UNC-Chapel Hill, [‡]AT&T Labs-Research, [§]Purdue University

Abstract

Critical network management applications increasingly demand fine-grained flow level measurements. However, current flow monitoring solutions are inadequate for many of these applications. In this paper, we present the design, implementation, and evaluation of CSAMP, a system-wide approach for flow monitoring. The design of CSAMP derives from three key ideas: flow sampling as a router primitive instead of uniform packet sampling; hash-based packet selection to achieve coordination without explicit communication; and a framework for distributing responsibilities across routers to achieve network-wide monitoring goals while respecting router resource constraints. We show that CSAMP achieves much greater monitoring coverage, better use of router resources, and enhanced ability to satisfy network-wide flow monitoring goals compared to existing solutions.

1 Introduction

Network operators routinely collect flow-level measurements to guide several network management applications. Traditionally, these measurements were used for customer accounting [9] and traffic engineering [13], which largely rely on aggregate traffic volume statistics. Today, however, flow monitoring assists several other critical network management tasks such as anomaly detection [19], identification of unwanted application traffic [6], and even forensic analysis [38], which need to identify and analyze as many distinct flows as possible. The main consequence of this trend is the increased need to obtain fine-grained flow measurements.

Yet, because of technological and resource constraints, modern routers cannot each record all packets or flows that pass through them. Instead, they rely on a variety of *sampling* techniques to selectively record as many packets as their CPU and memory resources allow. For example, most router vendors today implement uniform packet sampling (e.g., Netflow [5]); each router independently selects a packet with a sampling probability (typically between 0.001 and 0.01) and aggregates the selected packets into flow records. While sampling makes passive measurement technologically feasible (i.e., operate within the router constraints), the overall fidelity of flow-level measurements is reduced.

There is a fundamental disconnect between the increasing requirements of new network management applications and what current sampling techniques can provide. While router resources do scale with technological

advances, it is unlikely that this disconnect will disappear entirely, as networks continue to scale as well. We observe that part of this disconnect stems from a router-centric view of current measurement solutions. In today's networks, routers record flow measurements completely *independently* of each other, thus leading to redundant flow measurements and inefficient use of router resources.

We argue that a centralized system that coordinates monitoring responsibilities across different routers can significantly increase the flow monitoring capabilities of a network. Moreover, such a centralized system simplifies the process of specifying and realizing network-wide flow measurement objectives. We describe Coordinated Sampling (CSAMP), a system for coordinated flow monitoring within an Autonomous System (AS). CSAMP treats a network of routers *as a system to be managed in a coordinated fashion* to achieve specific measurement objectives. Our system consists of three design primitives:

- *Flow sampling*: CSAMP uses flow sampling [15] instead of traditional packet sampling to avoid the sampling biases against small flows—a feature of particular importance to the new spectrum of security applications. At the same time, flow sampling preserves the fidelity of traffic volume estimation and thus the accuracy of traditional traffic engineering applications.
- *Hash-based coordination*: CSAMP uses a hash-based selection primitive to eliminate duplicate measurements in the network. This allows different routers to monitor disjoint sets of flows without requiring explicit communication between routers, thus eliminating redundant and possibly ambiguous measurements across the network.
- *Network-wide optimization*: Finally, CSAMP uses an optimization framework to specify and satisfy network-wide monitoring objectives while respecting router resource constraints. The output of this optimization is then translated into per-router *sampling manifests* that specify the set of flows that each router is required to record.

We address several practical aspects in the design and implementation of CSAMP. We implement efficient algorithms for computing sampling manifests that scale to large tier-1 backbone networks with hundreds of routers. We provide practical solutions for handling multi-path routing and realistic changes in traffic patterns. We also implement a prototype using an off-the-shelf flow collection tool.

We demonstrate that CSAMP is fast enough to respond in real time to realistic network dynamics. Using network-wide evaluations on the Emulab testbed, we also show that CSAMP naturally balances the monitoring load across the network, thereby avoiding reporting hotspots. We evaluate the benefits of CSAMP over a wide range of network topologies. CSAMP observes more than twice as many flows compared with traditional uniform packet sampling, and is even more effective at achieving system-wide monitoring goals. For example, in the case of the minimum fractional flow coverage across all pairs of ingress-egress pairs, it provides significant improvement over other flow monitoring solutions. ISPs can derive several operational benefits from CSAMP, as it reduces the reporting bandwidth and the data management overheads caused by duplicated flow reports. We also show that CSAMP is robust with respect to errors in input data and realistic changes in traffic.

2 Related Work

The design of CSAMP as a centrally managed network-wide monitoring system is inspired by recent trends in network management. In particular, recent work has demonstrated the benefits of a network-wide approach for traffic engineering [13, 41] and network diagnosis [19, 20, 23]. Other recent proposals suggest that a centralized approach can significantly reduce management complexity and operating costs [1, 2, 14].

Despite the importance of network-wide flow monitoring, there have been few attempts in the past to design such systems. Most of the related work focuses on the single-router case and on providing incremental solutions to work around the limitations of uniform packet sampling. This includes work on adapting the packet sampling rate to changing traffic conditions [11, 17], tracking heavy-hitters [12], obtaining better traffic estimates from sampled measurements [9, 15], reducing the overall amount of measurement traffic [10], and data streaming algorithms for specific applications [18, 21].

Early work on network-wide monitoring has focused on the placement of monitors at appropriate locations to cover all routing paths using as few monitors as possible [4, 35]. The authors show that such a formulation is NP-hard, and propose greedy approximation algorithms. In contrast, CSAMP assumes a given set of monitoring locations along with their resource constraints and, therefore, is complementary to these approaches.

There are extensions to the monitor-placement problem in [35] to incorporate packet sampling. Cantieni et al. also consider a similar problem [3]. While the constrained optimization formulation in these problems shares some structural similarity to our approach in Section 4.2, the specific contexts in which these formulations are applied are different. First, CSAMP focuses on flow

sampling as opposed to packet sampling. By using flow sampling, CSAMP provides a generic flow measurement primitive that subsumes the specific traffic engineering applications that packet sampling (and the frameworks that rely on it) can support. Second, while it is reasonable to assume that the probability of a single packet being sampled multiple times across routers is negligible, this assumption is not valid in the context of flow-level monitoring. The probability of two routers sampling the same flow is high as flow sizes follow heavy-tailed distributions [7, 40]. Hence, CSAMP uses mechanisms to coordinate routers to avoid duplicate flow reporting.

To reduce duplicate measurements, Sharma and Byers [33] suggest the use of Bloom filters. While minimizing redundant measurements is a common high-level theme between CSAMP and their approach, our work differs on two significant fronts. First, CSAMP allows network operators to directly specify and satisfy network-wide objectives, explicitly taking into account (possibly heterogeneous) resource constraints on routers, while their approach does not. Second, CSAMP uses hash-based packet selection to implement coordination *without* explicit communication, while their approach requires every router to inform every other router about the set of flows it is monitoring.

Hash-based packet selection as a router-level primitive was suggested in Trajectory Sampling [8]. Trajectory Sampling assigns all routers in the network a *common* hash range. Each router in the network records the passage for all packets that fall in this common hash range. The recorded trajectories of the selected packets are then used for applications such as fault diagnosis. In contrast, CSAMP uses hash-based selection to achieve the opposite functionality: it assigns *disjoint* hash ranges across multiple routers so that different routers monitor different flows.

3 Motivation

We identify five criteria that a flow monitoring system should satisfy: (i) provide high flow coverage, (ii) minimize redundant reports, (iii) satisfy network-wide flow monitoring objectives (e.g., specifying some subsets of traffic as more important than others or ensuring fairness across different subsets of traffic), (iv) work within router resource constraints, and (v) be general enough to support a wide spectrum of flow monitoring applications. Table 1 shows a qualitative comparison of various flow monitoring solutions across these metrics.

Packet sampling implemented by routers today is inherently biased toward large flows, thus resulting in poor flow coverage. Thus, it does not satisfy the requirements of many classes of security applications [25]. In addition, this bias increases redundant flow reporting.

There exist solutions (e.g., [12, 18, 21]) that operate

	Uniform Packet Sampling (e.g. [3, 5])	Data Streaming Algorithms (e.g., [18, 21])	Heavy-hitter monitoring (e.g., [12])	Flow sampling (low-rate)	Flow sampling (high-rate)	CSAMP
High flow coverage	×	×	×	×	✓	✓
Avoiding redundant measurements	×	×	×	×	×	✓
Network-wide flow monitoring goals	×	×	×	×	×	✓
Operate within resource constraints	✓	✓	✓	✓	×	✓
Generality to support many applications	×	×	×	×	✓	✓

Table 1: Qualitative comparison across different deployment alternatives available to network operators.

efficiently within router resource constraints, but either lack generality across applications or, in fact, *reduce* flow coverage. For example, techniques for tracking flows with high packet counts (e.g., [10, 12]) are attractive single-router solutions for customer accounting and traffic engineering. However, they increase redundant monitoring across routers without increasing flow coverage.

Flow sampling is better than other solutions in terms of flow coverage and avoiding bias toward large flows. However, there is an inherent tradeoff between the flow coverage and router resources such as reporting bandwidth and load. Also, flow sampling fails to achieve network-wide objectives with sufficient fidelity.

As Table 1 shows, none of the existing solutions *simultaneously* satisfy all the criteria. To do so, we depart from the router-centric approach adopted by existing solutions and take a more system-wide approach. In the next section, we describe how CSAMP satisfies these goals by considering the routers in the network as a system to be managed in a coordinated fashion to achieve network-wide flow monitoring objectives.

4 Design

In this section, we present the design of the hash-based flow sampling primitive and the optimization engine used in CSAMP. In the following discussion, we assume the common 5-tuple (*srcIP*, *dstIP*, *srcport*, *dstport*, *protocol*) definition of an IP flow.

4.1 Router primitives

Hash-based flow sampling: Each router has a *sampling manifest* – a table of hash ranges indexed using a key. Upon receiving a packet, the router looks up the hash range using a key derived from the packet’s header fields. It computes the hash of the packet’s 5-tuple and samples the packet if the hash falls within the range obtained from the sampling manifest. In this case, the hash is used as an index into a table of flows that the router is currently monitoring. If the flow already exists in the table, it updates the byte and packet counters (and other statistics) for the flow. Otherwise it creates a new entry in the table.

The above approach implements flow sampling [15], since only those flows whose hash lies within the hash range are monitored. Essentially, we can treat the hash as a function that maps the input 5-tuple into a random value in the interval $[0, 1]$. Thus, the size of each hash

range determines the flow sampling rate of the router for each category of flows in the sampling manifest.

Flow sampling requires flow table lookups for each packet; the flow table, therefore, needs to be implemented in fast SRAM. Prior work has shown that maintaining counters in SRAM is feasible in many situations [12]. Even if flow counters in SRAM are not feasible, it is easy to add a packet sampling stage prior to flow sampling to make DRAM implementations possible [17]. For simplicity, however, we assume that the counters can fit in SRAM for the rest of the paper.

Coordination: If each router operates in isolation, i.e., independently sampling a subset of flows it observes, the resulting measurements from different routers are likely to contain duplicates. These duplicate measurements represent a waste of memory and reporting bandwidth on routers. In addition, processing duplicated flow reports incurs additional data management overheads.

Hash-based sampling enables a simple but powerful coordination strategy to avoid these duplicate measurements. Routers are configured to use the same hash function, but are assigned disjoint hash ranges so that the hash of any flow will match at most one router’s hash range. The sets of flows sampled by different routers will therefore not overlap. Importantly, assigning non-overlapping hash ranges achieves coordination *without* explicit communication. Routers can thus achieve coordinated tasks without complex distributed protocols.

4.2 Network-wide optimization

ISPs typically specify their network-wide goals in terms of *Origin-Destination (OD) pairs*, specified by the ingress and egress routers. To achieve flow monitoring goals specified in terms of OD-pairs, CSAMP’s optimization engine needs the traffic matrix (the number of flows per OD-pair) and routing information (the router-level path(s) per OD-pair), both of which are readily available to network operators [13, 41].

Assumptions and notation: We make two assumptions to simplify the discussion. First, we assume that the traffic matrix (number of IP flows per OD-pair) and routing information for the network are given exactly and that these change infrequently. Second, we assume that each OD-pair has a single router-level path. We relax these assumptions in Section 4.4 and Section 4.5.

Each OD-pair OD_i ($i = 1, \dots, M$) is characterized by its router-level path P_i and the number T_i of IP flows in a measurement interval (e.g., five minutes).

Each router R_j ($j = 1, \dots, N$) is constrained by two resources: memory (per-flow counters in SRAM) and bandwidth (for reporting flow records). (Because we assume that the flow counters are stored in SRAM, we do not model packet processing constraints [12].) We abstract these into a single resource constraint L_j , the number of flows router R_j can record and report in a given measurement interval.

Let d_{ij} denote the fraction of the IP flows of OD_i that router R_j samples. If R_j does not lie on path P_i , then the variable d_{ij} will not appear in the formulation. For $i = 1, \dots, M$, let C_i denote the fraction of flows on OD_i that is monitored.

Objective: We present a general framework that is flexible enough to support several possible flow monitoring objectives specified as (weighted) combinations of the different C_i values. As a concrete objective, we consider a hybrid measurement objective that maximizes the total flow-coverage across all OD-pairs ($\sum_i T_i \times C_i$) subject to ensuring the optimal minimum fractional coverage per OD-pair ($\min_i \{C_i\}$).

Problem *maxtotgivenfrac*(α):

$$\text{Maximize } \sum_i (T_i \times C_i), \text{ subject to}$$

$$\forall j, \quad \sum_{i: R_j \in P_i} (d_{ij} \times T_i) \leq L_j \quad (1)$$

$$\forall i, \quad C_i = \sum_{j: R_j \in P_i} d_{ij} \quad (2)$$

$$\forall i, \forall j, \quad d_{ij} \geq 0 \quad (3)$$

$$\forall i, \quad C_i \leq 1 \quad (4)$$

$$\forall i, \quad C_i \geq \alpha \quad (5)$$

We define a linear programming (LP) formulation that takes as a parameter α , the desired minimum fractional coverage per OD-pair. Given α , the LP maximizes the total flow coverage subject to ensuring that each OD-pair achieves a fractional coverage at least α , and that each router operates within its load constraint.

We briefly explain each of the constraints. (1) ensures that the number of flows that R_j is required to monitor does not exceed its resource constraint L_j . As we only consider sampling manifests in which the routers on P_i for OD_i will monitor distinct flows, (2) says that the fraction of traffic of OD_i that has been covered is simply the sum of the fractional coverages d_{ij} of the different routers on P_i . Because each C_i represents a fractional quantity we have the natural upper bound $C_i \leq 1$ in

(4). Since we want to guarantee that the fractional coverage on each OD-pair is greater than the desired minimum fractional coverage, we have the lower bound in (5). Since the d_{ij} define fractional coverages, they are constrained to be in the range $[0, 1]$; however, the constraints in (4) subsume the upper bound on each d_{ij} and we impose the non-zero constraints in (3).

To maximize the total coverage subject to achieving the highest possible minimum fractional coverage, we use a two-step approach. First, we obtain the optimal minimum fractional coverage by considering the problem of maximizing $\min_i \{C_i\}$ subject to constraints (1)–(4). Next, the value of *OptMinFrac* obtained from this optimization is used as the input α to *maxtotgivenfrac*.

The solution to the above two-step procedure, $d^* = \langle d_{ij}^* \rangle_{1 \leq i \leq M, 1 \leq j \leq N}$ provides a sampling strategy that maximizes the total flow coverage subject to achieving the optimal minimum fractional coverage per OD-pair.

4.3 Sampling manifests

The next step is to map the optimal solution into a *sampling manifest* for each router that specifies its monitoring responsibilities (Figure 1). The algorithm iterates over the M OD-pairs. For each OD_i , the variable *Range* is advanced in each iteration (i.e., per router) by the fractional coverage d_{ij}^* provided by the current router (lines 4 and 5 in Figure 1). This ensures that routers on the path P_i for OD_i are assigned disjoint ranges. Thus, no flows are monitored redundantly.

Once a router has received its sampling manifest, it implements the algorithm shown in Figure 2. For each packet it observes, the router first identifies the OD-pair. Next, it computes a hash on the flow headers (the IP 5-tuple) and checks if the hash value lies in the assigned hash range for the OD-pair (the function *HASH* returns a value in the range $[0, 1]$). That is, the key used for looking up the hash range (c.f., Section 4.1) is the flow's OD-pair. Each router maintains a *Flowtable* of the set of flows it is currently monitoring. If the packet has been selected, then the router either creates a new entry (if none exists) or updates the counters for the corresponding entry in the *Flowtable*.

4.4 Handling inaccurate traffic matrices

The discussion so far assumed that the traffic matrices are known and fixed. Traffic matrices are typically obtained using estimation techniques (e.g., [41, 42]) that may have estimation errors.

If the estimation errors are bounded, we scale the sampling strategy appropriately to ensure that the new scaled solution will operate within the router resource constraints and be near-optimal in comparison to an optimal solution for the true (but unknown) traffic matrix.

```

GENERATESAMPLINGMANIFEST( $d^* = \langle d_{ij}^* \rangle$ )
  //  $i$  ranges over all OD-pairs
  1 for  $i = 1, \dots, M$  do
  2    $Range \leftarrow 0$ 
  3   //  $j$  ranges over routers
  4   for  $j = 1, \dots, N$  do
  5      $HashRange(i, j) \leftarrow [Range, Range + d_{ij}^*)$ 
  6      $Range \leftarrow Range + d_{ij}^*$ 
  7    $\forall j, Manifest(j) \leftarrow \{ \langle i, HashRange(i, j) \rangle | d_{ij}^* > 0 \}$ 

```

Figure 1: Translating the optimal solution into a sampling manifest for each router

```

COORDSAMPROUTER( $pkt, Manifest$ )
  //  $Manifest = \langle i, HashRange(i, j) \rangle$ 
  1  $OD \leftarrow GETODPAIRID(pkt)$ 
  2 // HASH returns a value in  $[0, 1]$ 
  3  $h_{pkt} \leftarrow HASH(FLOWHEADER(pkt))$ 
  4 if  $h_{pkt} \in Hashrange(OD, j)$  then
  5   Create an entry in Flowtable if none exists
  6   Update byte and packet counters for the entry

```

Figure 2: Algorithm to implement coordinated sampling on router R_j

Suppose the estimation errors in the traffic matrix are bounded, i.e., if T_i and \hat{T}_i denote the estimated and actual traffic for OD_i respectively, then $\forall i, T_i \in [\hat{T}_i(1 - \epsilon), \hat{T}_i(1 + \epsilon)]$. Here, ϵ quantifies how much the estimated traffic matrix (i.e., our input data) differs with respect to the true traffic matrix. Suppose the optimal sampling strategy for $\hat{T} = \langle \hat{T}_i \rangle_{1 \leq i \leq M}$ is $\hat{d} = \langle \hat{d}_{ij} \rangle_{1 \leq i \leq M, 1 \leq j \leq N}$, and that the optimal sampling strategy for $T = \langle T_i \rangle_{1 \leq i \leq M}$ is $d^* = \langle d_{ij}^* \rangle_{1 \leq i \leq M, 1 \leq j \leq N}$.

A sampling strategy d is T -feasible if it satisfies conditions (1)–(4) for T . For a T -feasible strategy d , let $\beta(d, T) = \min_i \{C_i\}$ denote the minimum fractional coverage, and let $\gamma(d, T) = \sum_i T_i \times C_i = \sum_i T_i \times (\sum_j d_{ij})$ denote the total flow coverage. Setting $d'_{ij} = d_{ij}^*(1 - \epsilon)$, we can show that d' is \hat{T} -feasible, and¹

$$\begin{aligned} \beta(d', \hat{T}) &\geq \left(\frac{1 - \epsilon}{1 + \epsilon} \right) \beta(\hat{d}, \hat{T}) \\ \gamma(d', \hat{T}) &\geq \left(\frac{1 - \epsilon}{1 + \epsilon} \right)^2 \gamma(\hat{d}, \hat{T}). \end{aligned}$$

For example, with $\epsilon = 1\%$, using d' yields a worst case performance reduction of 2% in the minimum fractional coverage and 4% in the total coverage with respect to the optimal strategy \hat{d} .

¹For brevity, we do not show the full derivation of these results and refer the reader to the accompanying technical report [31].

4.5 Handling multiple paths per OD-pair

Next, we discuss a practical extension to incorporate multiple paths per OD-pair, for example using equal cost multi-path routing (ECMP).²

Given the routing and topology information, we can obtain the multiple routing paths for each OD-pair and can compute the number of flows routed across each of the multiple paths. Then, we treat each of the different paths as a distinct logical OD-pair with different individual traffic demands. As an example, suppose OD_i has two paths P_i^1 and P_i^2 . We treat P_i^1 and P_i^2 as independent OD-pairs with traffic values T_i^1 and T_i^2 . This means that we introduce additional d_{ij} variables in the formulation. In this example, in (1) we expand the term $d_{ij} \times T_i$ for router R_j to be $d_{ij}^1 \times T_i^1 + d_{ij}^2 \times T_i^2$ if R_j lies on both P_i^1 and P_i^2 .

However, when we specify the objective function and the sampling manifests, we merge these logical OD-pairs. In the above example, we would specify the network-wide objectives in terms of the total coverage for the OD_i , $C_i = C_i^1 + C_i^2$. This merging procedure also applies to the sampling manifests. For example, suppose R_j occurs on the two paths in the above example, and the optimal solution has values d_{ij}^1 and d_{ij}^2 corresponding to P_i^1 and P_i^2 . The sampling manifest simply specifies that R_j is responsible for a total fraction $d_{ij} = d_{ij}^1 + d_{ij}^2$ of the flows in OD_i .

5 System Architecture

Figure 3 depicts the overall architecture of CSAMP. The centralized optimization engine computes and disseminates sampling manifests based on the traffic matrix and routing information continuously measured in the network. This engine also assigns an identifier to every OD-pair and propagates this information to the ingress routers. The ingress routers determine the OD-pair and mark packets with the identifier. Each router uses the OD-pair identifier and its sampling manifest to decide if it should record a specific flow. In order to handle traffic dynamics, the optimization engine recalculates the traffic matrix periodically based on the observed flow reports to generate and distribute new sampling manifests. Such a centralized approach is consistent with the operating model of modern ISPs, where operators push out router configuration files (e.g., routing tables, ACLs) and collect information from the routers.

To complete the description of the CSAMP system, we describe the following mechanisms: 1) obtaining OD-pair information for packets; 2) responding to long- and short-term traffic dynamics; 3) managing memory re-

²ECMP-enabled routers make forwarding decisions on a per-IP-flow rather than on a per-packet basis. Thus, we need not be concerned with multiple packets from a single flow traversing different router-level paths.

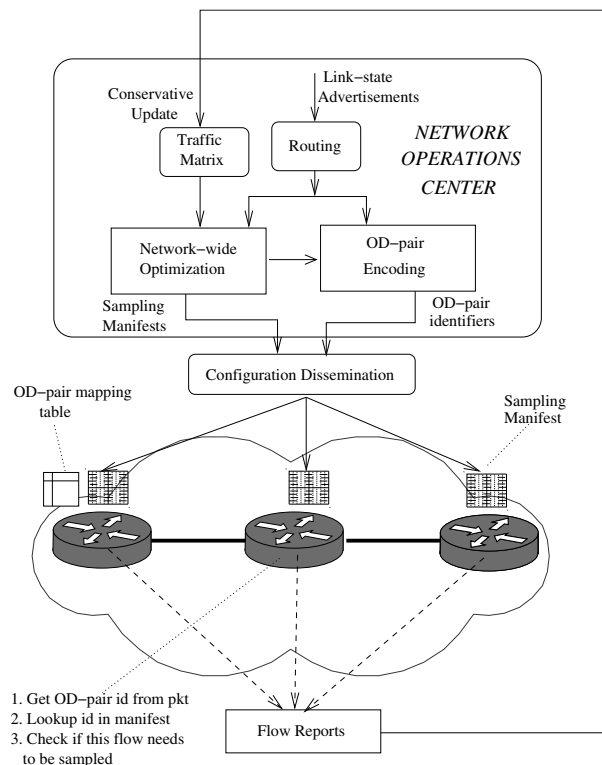


Figure 3: An overall view of the architecture of the CSAMP system. The optimization engine uses up-to-date traffic and routing information to compute and disseminate sampling manifests to routers.

sources on routers; 4) computing the sampling manifests efficiently; and 5) reacting to routing dynamics.

5.1 OD-pair identification

Each router, on observing a packet, must identify the OD-pair to which the packet belongs. There are prior approaches to infer the OD-pair for a given packet based on the source and destination IP addresses and routing information [13]. However, such information may not be immediately discernible to interior routers from their routing tables due to prefix aggregation. Ingress routers are in a better position to identify the appropriate egress when a packet enters the network using such techniques. Thus the ingress routers mark each packet header with the OD-pair identifier. Interior routers can subsequently extract this information. In practice, the OD-pair identifier can either be added to the IP-header or to the MPLS label stack. Note that the multi-path extension (Section 4.5) does not impose additional work on the ingress routers for OD-pair identification. In both the single-path and multi-path cases, an ingress router only needs to determine the egress router and the identifier for the ingress-egress pair, and need not distinguish between the different paths for each ingress-egress pair.

The identifier can be added to the IP-id field in a manner similar to other proposals that rely on packet marking (e.g., [22, 29, 39]). This 16-bit field allows assigning a unique identifier to each OD-pair in a network with up to 256 border routers (and 65,536 OD-pairs), which suffices for medium-sized networks. For larger ISPs, we use an additional encoding step to assign identifiers to OD-pairs so that there are no conflicts in the assignments. For example, OD_i and $OD_{i'}$ can be assigned the same identifier if P_i and $P_{i'}$ do not traverse a common router (and the same interfaces on that router) or, if they do, the common router is not assigned logging responsibility for one of them. We formulate this notion of non-conflicting OD-pairs as a graph coloring problem, and run a greedy coloring algorithm on the resulting conflict graph. Using this extension, the approach scales to larger ISPs (e.g., needing fewer than 10 bits to encode all OD-pairs for a network with 300 border routers). In the interest of space, we do not discuss this technique or the encoding results further.

While the above approach to retrofit OD-pair identifiers within the IP header requires some work, it is easier to add the OD-pair identifier as a static label in the MPLS label stack. In this case, the space required to specify OD-pair identifiers is not a serious concern.

5.2 Dealing with traffic dynamics

To ensure that the flow monitoring goals are achieved consistently over time, the optimization engine must be able to predict the traffic matrix to compute the sampling manifests. This prediction must take into account long-term variations in traffic matrices (e.g., diurnal trends), and also be able to respond to short-term dynamics (e.g., on the scale of a few minutes).

Long-term variations in traffic matrices typically arise from predictable time-of-day and day-of-week effects [28]. To handle these, we use historical traffic matrices as inputs to the optimization engine to compute the sampling strategy. For example, to compute the manifests for this week's Fri. 9am-10am period, we use the traffic matrix observed during the previous week's Fri. 9am-10am period.

The optimization engine also has to respond to less predictable short-term traffic variations. Using historical traffic matrices averaged over long periods (e.g., one week) runs the risk of *underfitting*; important structure present over shorter time scales is lost due to averaging. On the other hand, using historical traffic matrices over short periods (e.g., 5-minute intervals) may result in *overfitting*, unnecessarily incorporating details specific to the particular historical period in question.

To handle the long and short-term traffic dynamics, we take the following heuristic approach. Suppose we are interested in computing sampling manifests for every 5-

minute interval for the Fri. 9am-10am period of the current week. To avoid overfitting, we do not use the traffic matrices observed during the corresponding 5-minute intervals that make up the previous week's Fri. 9am-10am period. Instead, we take the (hourly) traffic matrix for the previous week's Fri. 9am-10am period, divide it by 12 (the number of 5-minute segments per hour), and use the resulting traffic matrix T^{old} as input data for computing the manifests for the first 5-minute period. At the end of this period, we collect flow data from each router and obtain the traffic matrix T^{obs} from the collected flow reports. (If the fractional coverage for OD_i with the current sampling strategy is C_i and x_i sampled flows are reported, then $T_i^{obs} = \frac{x_i}{C_i}$, i.e., normalizing the number of sampled flows by the total flow sampling rate.)

Given the observed traffic matrix for the current measurement period T^{obs} and the historical traffic matrix T^{old} , a new traffic matrix is computed using a *conservative update* policy. The resulting traffic matrix T^{new} is used as the input for obtaining the manifests for the next 5-minute period.

The conservative update policy works as follows. First, check if there are significant differences between the observed traffic matrix T^{obs} and the historical input data T^{old} . Let $\delta_i = \frac{|T_i^{obs} - T_i^{old}|}{T_i^{old}}$ denote the estimation error for OD_i . If δ_i exceeds a threshold Δ , then compute a new traffic matrix entry T_i^{new} , otherwise use T_i^{old} . If T_i^{obs} is greater than T_i^{old} , then set $T_i^{new} = T_i^{obs}$. If T_i^{obs} is smaller than T_i^{old} , check the resource utilization of the routers currently responsible for monitoring OD_i . If all these routers have residual resources available, set $T_i^{new} = T_i^{obs}$; otherwise set $T_i^{new} = T_i^{old}$.

The rationale behind this conservative update heuristic is that if a router runs out of resources, it may result in underestimating the new traffic on OD-pairs for which it is responsible (i.e., T^{obs} is an under-estimate of the actual traffic matrix). Updating T^{new} with T^{obs} for such OD-pairs is likely to cause a recurrence of the same overflow condition in the next 5-minute period. Instead, we err on the side of overestimating the traffic for each OD-pair. This ensures that the information obtained for the next period is reliable and can help make a better decision when computing manifests for subsequent intervals.

The only caveat is that this policy may provide lower flow coverage since it overestimates the total traffic volume. Our evaluations with real traffic traces (Section 6.3) show that this performance penalty is low and the heuristic provides near-optimal traffic coverage.

5.3 Flow records in SRAM

We assume that the flow table is maintained in (more expensive) SRAM. Thus, we need a compact representation of the flow record in memory, unlike Netflow [5] which maintains a 64-byte flow record in DRAM. We observe

that the entire flow record (the IP 5-tuple, the OD-pair identifier, and counters) need not actually be maintained in SRAM; only the flow counters (for byte and packet counts) need to be in SRAM. Thus, we can offload most of the flow fields to DRAM and retain only those relevant to the online computation: a four byte flow-hash (for flowtable lookups) and 32-bit counters for packets and bytes, requiring only 12 bytes of SRAM per flow record. To further reduce the SRAM required, we can use techniques for maintaining counters using a combination of SRAM and DRAM [43]. We defer a discussion of handling router memory exhaustion to Section 7.

5.4 Computing the optimal solution

In order to respond in near-real time to network dynamics, computing and disseminating the sampling manifests should require at most a few seconds. Unfortunately, the simple two-step approach in Section 4.2 requires a few hundreds of seconds on large ISP topologies and thus does not scale. We discovered that its bottleneck is the first step of solving the modified LP to find *OptMinFrac*.

To reduce the computation time we implement two optimizations. First, we use a binary search procedure to determine *OptMinFrac*. This was based on experimental evidence that solving the LP specified by *maxtotgivenfrac*(α) for a given α is faster than solving the LP to find *OptMinFrac*. Second, we use the insight that *maxtotgivenfrac*(α) can be formulated as a special instance of a MaxFlow problem. These optimizations reduce the time needed to compute the optimal sampling strategy to at most eleven seconds even on large tier-1 ISPs with more than 300 routers.

Binary search: The main idea is to use a binary search procedure over the value of α using the LP formulation *maxtotgivenfrac*(α). The procedure takes as input an error parameter ϵ and returns a feasible solution with a minimum fractional coverage α^* with the guarantee that $OptMinFrac - \alpha^* \leq \epsilon$. The search keeps track of α_{lower} , the smallest feasible value known (initially set to zero), and α_{upper} , the highest possible value (initially set to $\frac{\sum_j L_j}{\sum_i T_i}$). In each iteration, the lower and upper bounds are updated depending on whether the current value α is feasible or not and the current value α is updated to $\frac{\alpha_{lower} + \alpha_{upper}}{2}$. The search starts from $\alpha = \alpha_{upper}$, and stops if the gap $\alpha_{upper} - \alpha_{lower}$ is less than ϵ , and returns $\alpha^* = \alpha_{lower}$ at this stopping point.

Reformulation using MaxFlow: We formulate the LP *maxtotgivenfrac*(α) as an equivalent MaxFlow problem, specifically a variant of traditional MaxFlow problems that has additional lower-bound constraints on edge capacities. The intuition behind this optimization is that MaxFlow problems are typically more efficient to solve

than general LPs.

We construct the following (directed) graph $G = \langle V, E \rangle$. The set of vertices in G is

$$V = \{source, sink\} \cup \{od_i\}_{1 \leq i \leq M} \cup \{r_j\}_{1 \leq j \leq N}$$

Each od_i in the above graph corresponds to OD-pair OD_i in the network and each r_j in the graph corresponds to router R_j in the network.

The set of edges is $E = E_1 \cup E_2 \cup E_3$, where

$$E_1 = \{(source, od_i)\}_{1 \leq i \leq M}$$

$$E_2 = \{(r_j, sink)\}_{1 \leq j \leq N}$$

$$E_3 = \{(od_i, r_j)\}_{i,j: R_j \in P_i}$$

Let $f(x, y)$ denote the flow on the edge $(x, y) \in E$, and let $UB(x, y)$ and $LB(x, y)$ denote the upper-bound and lower-bound on edge capacities in G . Our objective is to maximize the flow F from $source$ to $sink$ subject to the following constraints.

$$\forall x, \left(\sum_y f(x, y) - \sum_y f(y, x) \right) = \begin{cases} F & x = source \\ -F & x = sink \\ 0 & \text{otherwise} \end{cases}$$

We specify lower and upper bounds on the flow on each edge as:

$$\forall x, \forall y, LB(x, y) \leq f(x, y) \leq UB(x, y)$$

The upper-bounds on the edge capacities are: (i) the edges from the $source$ to od_i have a maximum capacity equal to T_i (the traffic for OD-pair OD_i), and (ii) the edges from each r_j to the $sink$ have a maximum capacity equal to L_j (resource available on each router R_j).

$$UB((x, y)) = \begin{cases} T_i & x = source, y = od_i \\ L_j & x = r_j, y = sink \\ \infty & \text{otherwise} \end{cases}$$

We introduce lower bounds only on the edges from the $source$ to each od_i , indicating that each OD_i should have a fractional flow coverage at least α :

$$LB((x, y)) = \begin{cases} \alpha \times T_i & x = source, y = od_i \\ 0 & \text{otherwise} \end{cases}$$

We use the binary search procedure discussed earlier, but use this MaxFlow formulation to solve each iteration of the binary search instead of the LP formulation.

5.5 Handling routing changes

The CSAMP system receives real-time routing updates from a passive routing and topology monitor such as OSPF monitor [32]. Ideally, the optimization engine would recompute the sampling manifests for each routing update. However, recomputing and disseminating

sampling manifests to all routers for each routing update is expensive. Instead, the optimization engine uses a snapshot of the routing and topology information at the beginning of every measurement interval to compute and disseminate manifests for the next interval. This ensures that all topology changes are handled within at most two measurement intervals.

To respond more quickly to routing changes, the optimization engine can *precompute* sampling manifests for different failure scenarios in a given measurement cycle. Thus, if a routing change occurs, an appropriate sampling manifest corresponding to this scenario is already available. This precomputation reduces the latency of adapting to a given routing change to less than one measurement interval. Since it takes only a few seconds (e.g., 7 seconds for 300 routers and 60,000 OD-pairs) to compute a manifest on one CPU (Section 6.1), we can precompute manifests for all single router/link failure scenarios with a moderate (4-5 \times) level of parallelism. While precomputing manifests for multiple failure scenarios is difficult, such scenarios are also relatively rare.

5.6 Prototype implementation

Optimization engine: Our implementation of the algorithms for computing sampling manifests (Section 5.4) consists of 1500 lines of C/C++ code using the CPLEX callable library. The implementation is optimized for repeated computations with small changes to the input parameters, in that it carries state from one solution over to the next. Solvers like CPLEX typically reach a solution more quickly when starting “close” to a solution than when starting from scratch. Moreover, the solutions that result tend to have fewer changes to the preceding solutions than would solutions computed from scratch, which enables reconfigured manifests to be deployed with fewer or smaller messages. We implement this optimization for both our binary search algorithm and when recomputing sampling manifests in response to traffic and routing dynamics.

Flow collection: We implemented a CSAMP extension to the YAF flow collection tool.³ Our choice was motivated by our familiarity with YAF, its simplicity of implementation, and because it is a reference implementation for the IETF IPFIX working group. The extensions to YAF required 200 lines of additional code. The small code modification suggests that many current flow monitoring tools can be easily extended to realize the benefits of CSAMP. In our implementation, we use the BOB hash function recommended by Molina et al. [26].

6 Evaluation

We divide our evaluation into three parts. First, we demonstrate that the centralized optimization engine and

³<http://tools.netsa.cert.org/yaf>

the individual flow collection processes in CSAMP are scalable in Section 6.1. Second, we show the practical benefits that network operators can derive from CSAMP in Section 6.2. Finally, in Section 6.3, we show that the system can effectively handle realistic traffic dynamics.

In our experiments, we compare the performance of different sampling algorithms at a PoP-level granularity, i.e., treating each PoP as a “router” in the network model. We use PoP-level network topologies from educational backbones (Internet2 and GÉANT) and tier-1 ISP backbone topologies inferred by Rocketfuel [34]. We construct OD-pairs by considering all possible pairs of PoPs and use shortest-path routing to compute the PoP-level path per OD-pair. To obtain the shortest paths, we use publicly available static IS-IS weights for Internet2 and GÉANT and inferred link weights [24] for Rocketfuel-based topologies.

Topology (AS#)	PoPs	OD-pairs	Flows $\times 10^6$	Packets $\times 10^6$
NTT (2914)	70	4900	51	204
Level3 (3356)	63	3969	46	196
Sprint (1239)	52	2704	37	148
Telstra (1221)	44	1936	32	128
Tiscali (3257)	41	1681	32	218
GÉANT	22	484	16	64
Internet2	11	121	8	32

Table 2: Parameters for the experiments

Due to the lack of publicly available traffic matrices and aggregate traffic estimates for commercial ISPs, we take the following approach. We use a baseline traffic volume of 8 million IP flows for Internet2 (per 5-minute interval).⁴ For other topologies, we scale the total traffic by the number of PoPs in the topology (e.g., given that Internet2 has 11 PoPs, for Sprint with 52 PoPs the traffic is $\frac{52}{11} \times 8 = 37$ million flows). These values match reasonably well with traffic estimates reported for tier-1 ISPs. To model the structure of the traffic matrix, we first annotate PoP k with the population p_k of the city to which it is mapped. We then use a gravity model to obtain the traffic volume for each OD-pair [33]. In particular, we assume that the total traffic between PoPs k and k' is proportional to $p_k \times p_{k'}$. We assume that flow size (number of packets) is Pareto-distributed, i.e., $\Pr(\text{Flowsize} > x \text{ packets}) = (\frac{c}{x})^\gamma, x \geq c$ with $\gamma = 1.8$ and $c = 4$. (We use these as representative values; our results are similar across a range of flow size parameters.) Table 2 summarizes our evaluation setup.

6.1 Micro-benchmarks

In this section, we measure the performance of CSAMP along two dimensions – the cost of computing sampling

⁴The weekly aggregate traffic on Internet2 is roughly 175TB. Ignoring time-of-day effects, this translates into 0.08TB per 5-minute interval. Assuming an average flow size of 10KB, this translates into roughly 8 million flows.

manifests and the router overhead.

AS	PoP-level (secs)		Router-level (secs)	
	Bin-LP	Bin-MaxFlow	Bin-LP	Bin-MaxFlow
NTT	0.53	0.16	44.5	10.9
Level3	0.27	0.10	24.6	7.1
Sprint	0.01	0.08	17.9	4.8
Telstra	0.09	0.03	9.6	2.2
Tiscali	0.11	0.03	9.4	2.2
GÉANT	0.03	0.01	2.3	0.3
Internet2	0.01	0.005	0.20	0.14

Table 3: Time (in seconds) to compute the optimal sampling manifest for both PoP- and router-level topologies. Bin-LP refers to the binary search procedure without the MaxFlow optimization.

Computing sampling manifests: Table 3 shows the time taken to compute the sampling manifests on an Intel Xeon 2.80 GHz CPU machine for different topologies. For every PoP-level topology we considered, our optimization framework generates sampling manifests within one second, even with the basic LP formulation. Using the MaxFlow formulation reduces this further. On the largest PoP-level topology, NTT, with 70 PoPs, it takes only 160 ms to compute the sampling manifests with this optimization.

We also consider augmented router-level topologies constructed from PoP-level topologies by assuming that each PoP has four edge routers and one core router, with router-level OD-pairs between every pair of edge routers. To obtain the router-level traffic matrix, we split the inter-PoP traffic uniformly across the router-level OD-pairs constituting each PoP-level OD-pair.

Even with $5 \times$ as many routers and $16 \times$ as many OD-pairs as the PoP-level topologies, the worst case computation time is less than 11 seconds with the MaxFlow optimization. These results show that CSAMP can respond to network dynamics in near real-time, and that the optimization step is not a bottleneck.

Worst-case processing overhead: CSAMP imposes extra processing overhead per router to look up the OD-pair identifier in a sampling manifest and to compute a hash over the packet header. To quantify this overhead, we compare the throughput (on multiple offline packet traces) of running YAF in full flow capture mode, and running YAF with CSAMP configured to log every flow. Note that this configuration demonstrates the worst-case overhead because, in real deployments, a CSAMP instance would need to compute hashes only for packets belonging to OD-pairs that have been assigned to it, and update flow counters only for the packets it has selected. Even with this worst-case configuration the overhead of CSAMP is only 5% (not shown).

Network-wide evaluation using Emulab: We use Emulab [37] for a realistic network-wide evaluation of our prototype implementation. The test framework consists

of support code that (a) sets up network topologies; (b) configures and runs YAF instances per “router”; (c) generates offline packet traces for a given traffic matrix; and (d) runs real-time tests using the BitTwist⁵ packet replay engine with minor modifications. The only difference between the design in Section 4 and our Emulab setup is with respect to node configurations. In Section 4, sampling manifests are computed on a per-router basis, but YAF processes are instantiated on a per-interface basis. We map router-level manifests to interface-level manifests by assigning each router’s responsibilities across its ingress interfaces. For example, if R_j is assigned the responsibility to log OD_i , then this responsibility is assigned to the YAF process instantiated on the ingress interface for P_i on R_j .

We configure CSAMP in full-coverage mode, i.e., configured to capture all flows in the network (in our formulation this means setting the router resources such that $OptMinFrac = 1$). We also consider the alternative full coverage solution where each ingress router is configured to capture all traffic on incoming interfaces. The metric we compare is the normalized throughput of each YAF instance running in the emulated network. Let the total number of packets sent through the interface (in a fixed interval of 300 seconds) on which the YAF process is instantiated be $pkts_{actual}$. Suppose the YAF instance was able to process only $pkts_{processed}$ packets in the same time interval. Then the normalized throughput is defined as $\frac{pkts_{processed}}{pkts_{actual}}$. By definition, the normalized throughput can be at most 1.

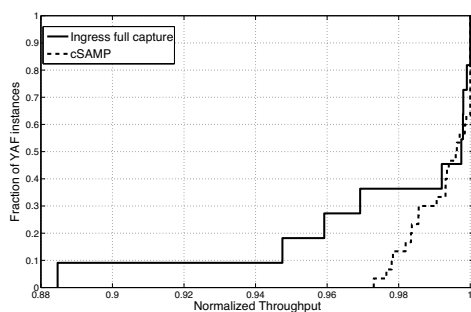


Figure 4: Comparing the CDF of normalized throughput per-interface across the entire network

Our test setup is unfair to CSAMP for two reasons. First, with a PoP-level topology, every ingress router is also a core router. Thus, there are no interior routers on which the monitoring load can be distributed. Second, to emulate a router processing packets on each interface, we instantiate multiple YAF processes on a single-CPU Emulab pc3000 node. In contrast, ingress flow capture

⁵<http://bittwist.sourceforge.net>

needs exactly one process per Emulab node. In reality, this processing would be either parallelized in hardware (offloaded to individual linecards), or on multiple CPUs per YAF process even in software implementations, or across multiple routers in router-level topologies.

Figure 4 shows the distribution of the normalized throughput values of each YAF instance in the emulated network. Despite the disadvantageous setup, the normalized packet processing throughput of CSAMP is higher. Given the 5% overhead due to hash computations mentioned before, this result might appear surprising. The better throughput of CSAMP is due to two reasons. First, each per-interface YAF instance incurs per-packet flow processing overheads (look up flowtable, update counters, etc.) only for the subset of flows assigned to it. Second, we implement a minor optimization that first checks whether the OD-pair (identified from IP-id field) for the packet is present in its sampling manifest, and computes a hash only if there is an entry for this OD-pair. We also repeated the experiment by doubling the total traffic volume, i.e., using 16 million flows instead of 8 million flows. The difference between the normalized throughputs is similar in this case as well. For example, the minimum throughput with ingress flow capture is only 85%, whereas for CSAMP the minimum normalized throughput is 93% (not shown). These results show that by distributing responsibilities across the network, CSAMP balances the monitoring load effectively.

6.2 Benefits of CSAMP

It is difficult to scale our evaluations to larger topologies using Emulab. Therefore, we implemented a custom packet-level network simulator (roughly 2500 lines of C++) to evaluate the performance of different sampling approaches. For all the sampling algorithms, the simulator uses the same network topology, OD traffic matrix, and IP flow-size distribution for consistent comparisons.

We consider two packet sampling alternatives: (i) uniform packet sampling with a sampling rate of 1-in-100 packets at all routers in the network, and (ii) uniform packet sampling at edge routers (this may reflect a feasible alternative for some ISPs [13]) with a packet sampling rate of 1-in-50 packets. We also consider two flow sampling variants: (iii) constant-rate flow sampling at all routers with a sampling rate of 1-in-100 flows, and (iv) maximal flow sampling in which the flow sampling rates are chosen such that each node maximally utilizes its available memory. In maximal flow sampling, the flow sampling rate for a router is $\min(1, \frac{l}{t})$, where l is the number of flow records it is provisioned to hold and t is the total number of flows it observes. Both constant-rate and maximal flow sampling alternatives are hypothetical; there are no implementations of either available in routers today. We consider them along with CSAMP

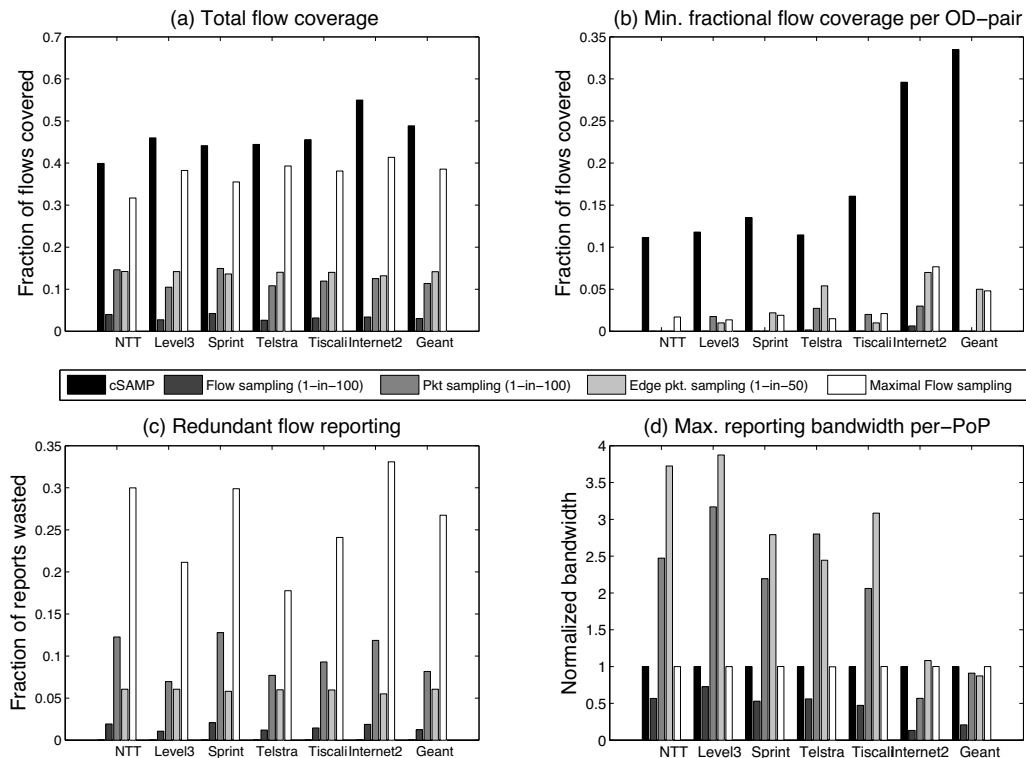


Figure 5: Comparing CSAMP with packet sampling and hypothetical flow sampling approaches

to evaluate different intermediate solutions in the overall design space, with current packet sampling approaches at one end of the spectrum and CSAMP at the other. The metrics we consider directly represent the criteria in Table 1.

CSAMP and the two flow sampling alternatives are constrained by the amount of SRAM on each router. We assume that each PoP in the network is provisioned to hold up to 400,000 flow records. Assuming roughly 5 routers per PoP, 10 interfaces per router, and 12 bytes per flow record, this requirement translates into $\frac{400,000 \times 12}{5 \times 10} = 96$ KB SRAM per linecard, which is well within the 8 MB technology limit (in 2004) suggested by Varghese [36]. (The total SRAM per linecard is shared across multiple router functions, but it is reasonable to allocate 1% of the SRAM for flow monitoring.) Since packet sampling alternatives primarily operate in DRAM, we use the methodology suggested by Estan and Varghese [12] and impose no memory restrictions on the routers. By assuming that packet sampling operates under no memory constraints, we provide it the best possible flow coverage (i.e., we underestimate the benefits of CSAMP).

Coverage benefits: Figure 5(a) compares the total flow coverage obtained with different sampling schemes for the various PoP-level topologies (Table 2). The total

flow coverage of CSAMP is $1.8\text{--}3.3\times$ that of the uniform packet sampling approaches for all the topologies considered. Doubling the sampling rate for edge-based uniform packet sampling only marginally improves flow coverage over all-router uniform packet sampling. Among the two flow sampling alternatives, constant rate flow sampling uses the available memory resources inefficiently, and the flow coverage is $9\text{--}16\times$ less than CSAMP. Maximal flow sampling uses the memory resources maximally, and therefore is the closest in performance. Even in this case, CSAMP provides $14\text{--}32\%$ better flow coverage. While this represents only a modest gain over maximal flow sampling, Figures 5(b) and 5(c) show that maximal flow sampling suffers from poor minimum fractional coverage and increases the amount of redundancy in flow reporting.

Figure 5(b) compares the minimum fractional coverage per OD-pair. CSAMP significantly outperforms all alternatives, including maximal flow sampling. This result shows a key strength of CSAMP to achieve network-wide flow coverage objectives, which other alternatives fail to provide. In addition, the different topologies vary significantly in the minimum fractional coverage, in comparison to the total coverage. For example, the minimum fractional coverage for Internet2 and GÉANT is significantly higher than other ASes even though the traffic

volumes in our simulations are scaled linearly with the number of PoPs. We attribute this to the unusually large diagonal and near-diagonal elements in a traffic matrix. For example, in the case of Telstra, the bias in the population distribution across PoPs is such that the top few densely populated PoPs (Sydney, Melbourne, and Los Angeles) account for more than 60% of the total traffic in the gravity-model based traffic matrix.

Reporting benefits: In Figure 5(c), we show the ratio of the number of *duplicate flow records* reported to the total number of distinct flow reports reported. The absence of CSAMP in Figure 5(c) is because of the assignment of non-overlapping hash-ranges to avoid duplicate monitoring. Constant rate flow sampling has little duplication, but it provides very low flow coverage. Uniform packet sampling can result in up to 14% duplicate reports. Edge-based packet sampling can alleviate this waste to some extent by avoiding redundant reporting from transit routers. Maximal flow sampling incurs the largest amount of duplicate flow reports (as high as 33%).

Figure 5(d) shows the *maximum reporting bandwidth* across all PoPs. We normalize the reporting bandwidth by the bandwidth required for CSAMP. The reporting bandwidth for CSAMP and flow sampling is bounded by the amount of memory that the routers are provisioned with; memory relates directly to the number of flow-records that a router needs to export. The normalized load for uniform packet sampling can be as high as four. Thus CSAMP has the added benefit of avoiding reporting hotspots unlike traditional packet sampling approaches.

Summary of benefits: CSAMP significantly outperforms traditional packet sampling approaches on all four metrics. Unlike constant rate flow sampling, CSAMP efficiently leverages the available memory resources. While maximal flow sampling can partially realize the benefits in terms of total flow coverage, it has poor performance with respect to the minimum fractional flow coverage and the number of duplicated flow reports. Also, as network operators provision routers to obtain greater flow coverage, this bandwidth overhead due to duplicate flow reports will increase.

6.3 Robustness properties

To evaluate the robustness of our approach to realistic traffic changes, we consider a two-week snapshot (Dec 1–14, 2006) of (packet sampled) flow data from Internet2. We map each flow entry to the corresponding network ingress and egress points using the technique outlined by Feldmann et al. [13].⁶ We assume that there are no routing changes in the network, and that the sampled flow records represent the actual traffic in the network.

⁶Since IP-addresses are anonymized by zero-ing out the last 11 bits, there is some ambiguity in egress resolution. However, this does not introduce a significant bias as less than 3% of the flows are affected.

(Since CSAMP does not suffer from flow size biases there is no need to renormalize the flow sizes by the packet sampling rate.) For this evaluation, we scale down the per-PoP memory to 50,000 flow records. (Due to packet sampling, the dataset contains fewer unique flows than the estimate in Table 2.)

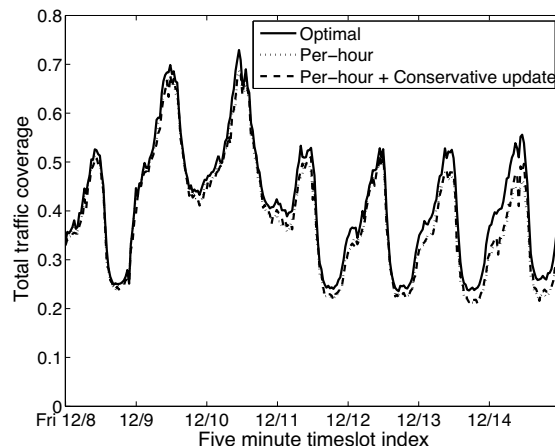


Figure 6: Comparing total traffic coverage vs. the optimal solution

Figure 6 compares the total flow coverage using our approach for handling traffic dynamics (Section 5.2) with the optimal total flow coverage (i.e., if we use the actual traffic matrix instead of the estimated traffic matrix to compute manifests). As expected, the optimal flow coverage exhibits time-of-day and day-of-week effects. For example, during the weekend, the coverage is around 70% while on the weekdays the coverage is typically in the 20-50% range. The result confirms that relying on traffic matrices that are based on hourly averages from the previous week gives near-optimal total flow coverage and represents a time scale of practical interest that avoids both overfitting and underfitting (Section 5.2). Using more coarse-grained historical information (e.g., daily or weekly averages) gives sub-optimal coverage (not shown). Figure 6 also shows that even though the conservative update heuristic (Section 5.2) overestimates the traffic matrix, the performance penalty arising from this overestimation is negligible.

Figure 7 shows that using the per-hour historical estimates alone performs poorly compared to the optimal minimum fractional coverage. This is primarily because of short-term variations that the historical traffic matrices cannot account for. The conservative update heuristic significantly improves the performance in this case and achieves near-optimal performance. These results demonstrate that our approach of using per-hour historical traffic matrices combined with a conservative update heuristic is robust to realistic traffic dynamics.

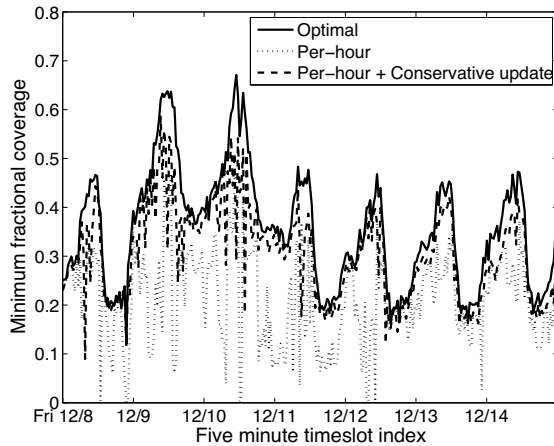


Figure 7: Comparing the minimum fractional coverage vs. the optimal solution

7 Discussion and Future Work

Reliance on OD-pair identifiers: A key limitation of our design is the reliance on OD-pair identifiers. This imposes two requirements: (i) modifications to packet headers, and (ii) upgrades to border routers to compute the egress router [13] for each packet. While this assumption simplifies our design, an interesting question is whether it is possible to realize the benefits of a CSAMP-like framework even when routers’ sampling decisions are based only on local information.

Router memory exhaustion: Despite factoring in the router memory constraints into the optimization framework, a router’s flow memory might be exhausted due to traffic dynamics. In our current prototype, we choose not to evict flow records already in the flow memory, but instead stop creating new flow records until the end of the measurement cycle. The conservative update heuristic (Section 5.2) will ensure that the traffic demands for the particular OD-pairs that caused the discrepancy are updated appropriately in the next measurement cycle.

In general, however, more sophisticated eviction strategies might be required to prevent unfairness within a given measurement cycle under adversarial traffic conditions. For example, one such strategy could be to allocate the available flow memory across all OD-pairs in proportion to their hash ranges and evict flows only from those OD-pairs that exceed their allotted share. While this approach appears plausible at first glance, it has the side effect that traffic matrices will not be updated properly to reflect traffic dynamics. Thus, it is important to jointly devise the eviction and the traffic matrix update strategies to prevent short-term unfairness, handle potential adversarial traffic conditions, and minimize the error in estimating traffic matrices. We intend to pursue such

strategies as part of future work.

Transient conditions inducing loss of flow coverage or duplication: A loss in flow coverage can occur if a router that has been assigned a hash range for an OD-pair no longer sees any traffic for that OD-pair due to a routing change. Routing changes will not cause any duplication if the OD-pair identifiers are globally unique. However, if we encode OD-pair identifiers without unique assignments (see Section 5.1), then routing changes could result in duplication due to OD-pair identifier aliasing. Also, due to differences in the time for new configurations to be disseminated to different routers, there is a small amount of time during which routers may be in inconsistent sampling configurations resulting in some duplication or loss.

Applications of CSAMP: CSAMP provides an efficient flow monitoring infrastructure that can aid and enable many new traffic monitoring applications (e.g., [6, 16, 19, 30, 38]). As an example application that can benefit from better flow coverage, we explored the possibility of uncovering botnet-like communication structure in the network [27]. We use flow-level data from Internet2 and inject 1,000 synthetically crafted single-packet flows into the original trace, simulating botnet command-and-control traffic. CSAMP uncovers $12\times$ (on average) more botnet flows compared to uniform packet sampling. We also confirmed that CSAMP provides comparable or better fidelity compared to uniform packet sampling for traditional traffic engineering applications such as traffic matrix estimation.

8 Conclusions

Flow-level monitoring is an integral part of the suite of network management applications used by network operators today. Existing solutions, however, focus on incrementally improving single-router sampling algorithms and fail to meet the increasing demands for fine-grained flow-level measurements. To meet these growing demands, we argue the need for a system-wide rather than router-centric approach for flow monitoring.

We presented CSAMP, a system that takes a network-wide approach to flow monitoring. Compared to current solutions, CSAMP provides higher flow coverage, achieves fine-grained network-wide flow coverage goals, efficiently leverages available monitoring capacity and minimizes redundant measurements, and naturally load balances responsibilities to avoid hotspots. We also demonstrated that our system is practical: it scales to large tier-1 backbone networks, it is robust to realistic network dynamics, and it provides a flexible framework to accommodate complex policies and objectives.

Acknowledgments

We thank Daniel Golovin for suggesting the scaling argument used in Section 4.4, Vineet Goyal and Anupam Gupta for insightful discussions on algorithmic issues, and Steve Gribble for shepherding the final version of the paper. This work was supported in part by NSF awards CNS-0326472, CNS-0433540, CNS-0619525, and ANI-0331653. Ramana Kompella thanks Cisco Systems for their support.

References

- [1] BALLANI, H., AND FRANCIS, P. CONMan: A Step Towards Network Manageability. In *Proc. of ACM SIGCOMM* (2007).
- [2] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, J. Design and implementation of a Routing Control Platform. In *Proc. of NSDI* (2005).
- [3] CANTIENI, G. R., IANNACONE, G., BARAKAT, C., DIOT, C., AND THIRAN, P. Reformulating the Monitor Placement problem: Optimal Network-Wide Sampling. In *Proc. of CoNeXT* (2006).
- [4] CHADET, C., FLEURY, E., LASSOUS, I., HERVÉ, AND VOGÉ, M.-E. Optimal Positioning of Active and Passive Monitoring Devices. In *Proc. of CoNeXT* (2005).
- [5] CLAISE, B. Cisco Systems NetFlow Services Export Version 9. RFC 3954.
- [6] COLLINS, M. P., AND REITER, M. K. Finding Peer-to-Peer File-sharing using Coarse Network Behaviors. In *Proc. of ESORICS* (2006).
- [7] DREGER, H., FELDMANN, A., KRISHNAMURTHY, B., WALLERICH, J., AND WILLINGER, W. A Methodology for Studying Persistency Aspects of Internet Flows. *ACM SIGCOMM CCR* 35, 2 (Apr. 2005).
- [8] DUFFIELD, N., AND GROSSGLAUSER, M. Trajectory Sampling for Direct Traffic Observation. In *Proc. of ACM SIGCOMM* (2001).
- [9] DUFFIELD, N., LUND, C., AND THORUP, M. Charging from sampled network usage. In *Proc. of IMW* (2001).
- [10] DUFFIELD, N., LUND, C., AND THORUP, M. Learn more, sample less: Control of volume and variance in network measurement. *IEEE Transactions in Information Theory* 51, 5 (2005), 1756–1775.
- [11] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a Better NetFlow. In *Proc. of ACM SIGCOMM* (2004).
- [12] ESTAN, C., AND VARGHESE, G. New Directions in Traffic Measurement and Accounting. In *Proc. of ACM SIGCOMM* (2002).
- [13] FELDMANN, A., GREENBERG, A. G., LUND, C., REINGOLD, N., REXFORD, J., AND TRUE, F. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. of ACM SIGCOMM* (2000).
- [14] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MEYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR* 35, 5 (Oct. 2005).
- [15] HOHN, N., AND VEITCH, D. Inverting Sampled Traffic. In *Proc. of IMC* (2003).
- [16] KARAGIANNIS, T., PAPAGIANNAKI, D., AND FALOUTSOS, M. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. of ACM SIGCOMM* (2005).
- [17] KOMPPELLA, R., AND ESTAN, C. The Power of Slicing in Internet Flow Measurement. In *Proc. of IMC* (2005).
- [18] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Distribution. In *Proc. of ACM SIGMETRICS* (2004).
- [19] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing Network-Wide Traffic Anomalies. In *Proc. of ACM SIGCOMM* (2004).
- [20] LAKHINA, A., PAPAGIANNAKI, K., CROVELLA, M., DIOT, C., KOLACZYK, E., AND TAFT, N. Structural Analysis of Network Traffic Flows. In *Proc. of ACM SIGMETRICS* (2004).
- [21] LALL, A., SEKAR, V., XU, J., OGIHARA, M., AND ZHANG, H. Data Streaming Algorithms for Estimating Entropy of Network Traffic. In *Proc. of ACM SIGMETRICS* (2006).
- [22] LI, J., SUNG, M., XU, J., LI, L., AND ZHAO, Q. Large-scale IP Traceback in High-speed Internet: Practical Techniques and Theoretical Foundation. In *Proc. of IEEE Symposium of Security and Privacy* (2004).
- [23] LI, X., BIAN, F., ZHANG, H., DIOT, C., GOVINDAN, R., HONG, W., AND IANNACONE, G. MIND: A Distributed Multidimensional Indexing for Network Diagnosis. In *Proc. of IEEE INFOCOM* (2006).
- [24] MAHAJAN, R., SPRING, N., WETHERALL, D., AND ANDERSON, T. Inferring Link Weights using End-to-End Measurements. In *Proc. of IMW* (2002).
- [25] MAI, J., CHUAH, C.-N., SRIDHARAN, A., YE, T., AND ZANG, H. Is Sampled Data Sufficient for Anomaly Detection? In *Proc. of IMC* (2006).
- [26] MOLINA, M., NICCOLINI, S., AND DUFFIELD, N. A Comparative Experimental Study of Hash Functions Applied to Packet Sampling. In *Proc. of International Teletraffic Congress (ITC)* (2005).
- [27] RAMACHANDRAN, A., SEETHARAMAN, S., FEAMSTER, N., AND VAZIRANI, V. Building a Better Mousetrap. Georgia Tech Technical Report, GIT-CSS-07-01, 2007.
- [28] ROUGHAN, M., GREENBERG, A., KALMANEK, C., RUMSEWICZ, M., YATES, J., AND ZHANG, Y. Experience in Measuring Internet Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *Proc. of International Teletraffic Congress (ITC)* (2003).
- [29] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Practical Network Support for IP Traceback. In *Proc. of ACM SIGCOMM* (2000).
- [30] SEKAR, V., DUFFIELD, N., VAN DER MERWE, K., SPATSCHECK, O., AND ZHANG, H. LADS: Large-scale Automated DDoS Detection System. In *Proc. of USENIX ATC* (2006).
- [31] SEKAR, V., REITER, M. K., WILLINGER, W., AND ZHANG, H. Coordinated Sampling: An Efficient Network-wide Approach for Flow Monitoring. Technical Report, CMU-CS-07-139, Computer Science Dept, Carnegie Mellon University, 2007.
- [32] SHAIKH, A., AND GREENBERG, A. OSPF Monitoring: Architecture, Design and Deployment Experience. In *Proc. of NSDI* (2004).
- [33] SHARMA, M. R., AND BYERS, J. W. Scalable Coordination Techniques for Distributed Network Monitoring. In *Proc. of PAM* (2005).
- [34] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM* (2002).
- [35] SUH, K., GUO, Y., KUROSE, J., AND TOWSLEY, D. Locating Network Monitors: Complexity, heuristics and coverage. In *Proc. of IEEE INFOCOM* (2005).
- [36] VARGHESE, G. *Network Algorithms*. Morgan Kaufman, 2005.
- [37] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI* (2002).
- [38] XIE, Y., SEKAR, V., MALTZ, D. A., REITER, M. K., AND ZHANG, H. Worm Origin Identification Using Random Moonwalks. In *Proc. of IEEE Symposium on Security and Privacy* (2005).
- [39] YANG, X., WETHERALL, D., AND ANDERSON, T. A DoS-limiting Network Architecture. In *Proc. of ACM SIGCOMM* (2005).
- [40] ZHANG, Y., BRESLAU, L., PAXSON, V., AND SHENKER, S. On the Characteristics and Origins of Internet Flow Rates. In *Proc. of ACM SIGCOMM* (2002).
- [41] ZHANG, Y., ROUGHAN, M., DUFFIELD, N., AND GREENBERG, A. Fast Accurate Computation of Large-scale IP Traffic Matrices from Link Loads. In *Proc. of ACM SIGMETRICS* (2003).
- [42] ZHAO, Q., GE, Z., WANG, J., AND XU, J. Robust Traffic Matrix Estimation with Imperfect Information: Making use of Multiple Data Sources. In *Proc. of ACM SIGMETRICS* (2006).
- [43] ZHAO, Q., XU, J., AND LIU, Z. Design of a novel statistics counter architecture with optimal space and time efficiency. In *Proc. of ACM SIGMETRICS* (2006).

Studying Black Holes in the Internet with Hubble

Ethan Katz-Bassett* Harsha V. Madhyastha* John P. John* Arvind Krishnamurthy*
David Wetherall† Thomas Anderson*

Abstract

We present *Hubble*, a system that operates continuously to find Internet reachability problems in which routes exist to a destination but packets are unable to reach the destination. *Hubble* monitors at a 15 minute granularity the data-path to prefixes that cover 89% of the Internet's edge address space. Key enabling techniques include a hybrid passive/active monitoring approach and the synthesis of multiple information sources that include historical data.

With these techniques, we estimate that *Hubble* discovers 85% of the reachability problems that would be found with a pervasive probing approach, while issuing only 5.5% as many probes. We also present the results of a three week study conducted with *Hubble*. We find that the extent of reachability problems, both in number and duration, is much greater than we expected, with problems persisting for hours and even days, and many of the problems do not correlate with BGP updates. In many cases, a multi-homed AS is reachable through one provider, but probes through another terminate; using spoofed packets, we isolated the direction of failure in 84% of cases we analyzed and found all problems to be exclusively on the forward path from the provider to the destination. A snapshot of the problems *Hubble* is currently monitoring can be found at <http://hubble.cs.washington.edu>.

1 Introduction

Global reachability – when every address is reachable from every other address – is the most basic goal of the Internet. It was specified as a top priority in the original design of the Internet protocols, ahead of high performance or good quality of service, with the philosophy that “there is only one failure, and it is complete partition” [4]. Today, middleboxes such as NATs complicate this picture by artificially restricting connectivity to addresses within some customer networks. Yet within the default-free core of the Internet, it should be the case that if there is a working physical path that is policy-compliant, then there will be a valid BGP path, and if there is a valid BGP path, then traffic will reach the destination. However, this is not always the case in practice; traffic may disappear into *black holes* and consistently fail to reach the destination. Outages that are not tran-

sient are problematic, as an operator generally has little visibility into other ASes to discern the nature of an outage and little ability to check if the problem exists from other vantage points. For example, black holes are a recurring theme on the Outages [28] and NANOG [29] mailing lists [7], with users asking whether others can reach their prefixes, or posting when they are unable to reach certain destinations, to ask if others see the same problem or know the cause.

Internet operations would thus benefit from having a system that automatically detects reachability problems and aids operators in locating the network entity (an AS, a router) responsible for the problem. Previous systems addressed aspects of this goal in different ways. A number of systems monitored reachability status in real-time, but within contexts that are narrower than the whole Internet, such as a testbed [23, 1, 6, 12], an autonomous system [34, 24], or a particular distributed system's clients [35]. Other systems such as *iPlane* [20] have broad and continuous Internet coverage but, being designed for other purposes, monitor at too infrequent a rate to provide real-time fault diagnosis. Still another body of work detects certain reachability issues in real-time at the Internet scale by passively monitoring BGP feeds [8, 34, 3]. But these techniques isolate anomalies at the level of autonomous systems and are thus too coarse-grained from the perspective of a network operator. More importantly, as our data will show, relying on BGP feeds alone is insufficient because the existence of a route does not imply reachability; BGP acts as a control plane to establish routes for the data plane on which Internet traffic flows, and connectivity problems that do not present themselves as events on the monitored control plane will evade such systems.

Our goal is to construct a system that can identify Internet reachability problems over the global Internet in real-time and locate the likely sources of problems. We call our system *Hubble* and focus initially on reachability problems, though we believe that the principles of our system extend to other data-plane problems. The major challenge in building *Hubble* is that of scale: how can we provide spatial and temporal coverage that scales to the global Internet, monitoring the data-plane from all vantages to all destinations, without requiring a prohibitive number of measurement probes.

In this paper, we describe the design and evaluation of *Hubble*. To identify potential problems, the system mon-

*Dept. of Computer Science, Univ. of Washington, Seattle.

†Univ. of Washington and Intel Research.

itors BGP feeds and continuously pings prefixes across the Internet from distributed PlanetLab sites. It then uses traceroutes and other probes from the sites to collect details about identified problems and uses a central repository of current and historical data to pinpoint where packets are being lost. We show that it monitors 89% of the Internet's edge prefix address space with a 15-minute time granularity and discovers 85% of the reachability issues that would be identified by a pervasive probing approach (in which all vantage points traceroute those same prefixes every 15 minutes), while issuing only 5.5% as many probes. We believe *Hubble* to be the first real-time tool to identify reachability problems on this scale across the Internet. The next largest system of which we are aware, PlanetSeer, covered half as many ASes in a 3-month study and monitored paths only for the small fraction of time that they were in use by clients [35]. *Hubble* has been running continuously since mid-September, 2007, identifying over 640,000 black holes and reachability problems in its first 4 months of operation. It also performs analysis using fine-grained real-time and historical probes to classify most problems, a step towards diagnosis for operators.

Hubble relies on two high-level techniques:

Hybrid monitoring: *Hubble* uses a hybrid passive/active monitoring approach to intelligently identify target prefixes likely to be experiencing problems. The approach combines passive monitoring of BGP feeds for the entire Internet with active monitoring of most of the Internet's edge. The two monitoring subsystems trigger distributed active probes when they suspect problems. Currently, they identify targets that might not be globally reachable, but in the future we plan to also look at performance issues such as latency. The hybrid approach allows *Hubble* to monitor the entire Internet while still providing router-level probe information from diverse vantage points at a reasonably fast pace during problems.

Synthesis of multiple information sources: In order to provide as much detail on problems as possible, *Hubble* combines multiple sources of information. For example, *Hubble* maintains historical records of successful traceroutes from its vantage points to destinations across the Internet and monitors the liveness of routers along these routes. When it finds that one of its vantage points is unable to reach a destination, it compares current router-level probe data from that site to its historical data and to probes from other sites, to determine the extent and possible location of the problem.

We also present observations on Internet reachability made from three weeks of *Hubble* data. We found reachability problems to be more common, widespread and longer lasting than we had expected. Over three weeks, we identified more than 31,000 reachability problems involving more than 10,000 distinct prefixes. While

many problems resolved within one hour, 10% persisted for more than one day. Many of the problems involved partial reachability, in which some vantage points can reach a prefix while others cannot, even though a working physical path demonstrably exists. This included cases in which destination prefixes with multi-homed origin ASes were reachable through one of the origin's providers and not another. It suggests that edge networks do not always get fault tolerance through multi-homing. Finally, we observed that many Internet reachability problems were not visible as events at commonly used BGP monitors. That is, BGP monitoring alone is not sufficient to discover the majority of problems.

The rest of this paper is organized as follows. We define the reachability problem in Section 2. In Section 3, we describe the design of *Hubble*. We present an evaluation of *Hubble* in Section 4 and use it to study Internet reachability in Section 5. Related work is given in Section 6 and we conclude in Section 7.

2 Problem

In this section, we present necessary background on Internet routing, then define the reachability problems we study.

2.1 Background

An autonomous system (AS) on the Internet is a collection of routers and networks that presents a common routing view to the rest of the Internet. Routes on the Internet are determined on a per-prefix basis, with the prefix comprising all IP addresses with p as their first n bits typically written as p/n . ASes exchange routes using the routing protocol BGP, with an AS announcing to its neighbor its ability to route to a particular prefix by giving the AS path it plans to use. An origin AS for a prefix is the first AS on an AS path to that prefix, and a multi-homed AS is one with multiple provider ASes.

2.2 Defining reachability problems

We are interested in reachability problems with four characteristics:

- *Routeable prefix.* We ignore cases in which we do not expect the prefix to be reachable, either because the prefix has never been reachable or the prefix has been completely withdrawn from BGP tables. BGP monitoring easily detects these problems already.
- *Persistent.* Although we may happen to detect short term route failures, such as those experienced during BGP convergence [18, 33, 16], our focus is on detecting persistent issues. We consider only problems that persist through 2 rounds of quarter-hourly probes.
- *Not simply end-system or end-network failures.* We are primarily interested in problems in which the Internet's routing infrastructure fails to provide connec-

tivity along advertised AS paths, rather than problems in which the traffic traverses the AS path, but the specific destination or prefix happens to be down. As such, though we also track whether probes reach the destination or its prefix, we make judgments on reachability problems based on connectivity to the origin AS.

- *Not simply source problems.* We are concerned with the reachability of destinations on an Internet-scale, rather than problems caused only by issues near one of our sources. In a study of four months of daily traceroutes from 30 PlanetLab vantage points to 110,000 prefixes, we found that most of the time, all but a few of the 30 traceroutes reached the prefix's origin AS. If less than 90% reached, though, it was likely the problems were more widespread; in half those cases, at least half of the traceroutes failed to reach. We use this value as our conservative threshold: if at least 90% of probes reach the origin AS of a prefix, then we assume that any probes that did not reach may have experienced congestion or problems near the source, and we ignore the issue.

For this paper, we concern ourselves primarily with reachability problems that fit these criteria. We use 90% reachability to define if a prefix is experiencing a *reachability problem*. We define a *reachability event* as the period starting when a prefix begins to experience a reachability problem and concluding when its reachability increases to 90% or higher. Such problems are often referred to as *black holes*, but we have found the term used in varying ways; instead, we use the term *reachability event* to refer to a network anomaly manifesting as a period in which a prefix experiences reachability problems.

3 Hubble Design and Architecture

Hubble attempts to discover and track reachability problems, as well as classify the problems in real-time as they occur. We base the classification on topological characteristics meant to aid diagnosis, e.g., is all of the destination traffic through a given AS affected, or only through a particular router? Is the failure related to a path change, or is it on a path that previously worked?

3.1 Goals

We seek to build a system that can provide information about ongoing reachability problems in the Internet. We hope that our system will be helpful for operators in identifying and diagnosing problems, so our system should aid in localizing the problem. It could also be used as a critical building block for overlay detouring services seeking to provide uninterrupted service between arbitrary end-hosts in the Internet. Given these potential applications, we require the system design to be driven by the following requirements.

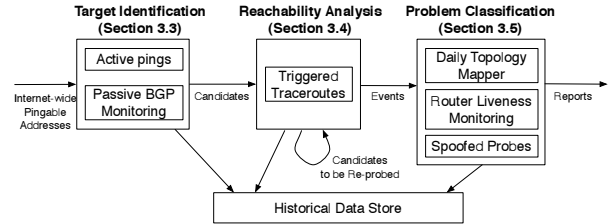


Figure 1: Diagram of the Hubble architecture.

Real-time and Continuous Information: The system should provide up-to-date and continuous information, so network operators and distributed services can quickly react to ongoing problems.

Data-plane focused: We desire a system that detects data reachability problems, regardless of whether or not they appear as BGP events. The Internet is intended to deliver data, and a BGP path on top of a broken data plane is useless. The only way to absolutely discern whether the data plane is functioning and packets can reach a destination is to send traffic, such as a traceroute, towards that destination.

Global-scale: The modern Internet is globally pervasive, and we desire a system that can monitor reachability problems for destinations of the entire Internet simultaneously, identifying most or all long-lasting reachability problems that occur. The probing and computation requirements must feasibly scale.

Light measurement traffic overhead: We intend our system to monitor areas of the Internet experiencing problems, and, in doing so, we do not want to exacerbate the problems. Our system relies on routers configured to respond to measurement probes. For these reasons, we desire a system that introduces as little measurement traffic as possible in its monitoring.

3.2 Overview of Measurement Components

As depicted in Figure 1, *Hubble* combines multiple types of measurements into four main components to identify and classify problems: pingable address discovery to decide what to monitor (not shown in figure); active ping monitoring and passive BGP monitoring to identify potential problem prefixes as targets for reachability assessment; triggered traceroutes to assess reachability and monitor reachability problems; and daily topology mapping, router liveness monitoring, and spoofed probes, combined with the same triggered traceroutes, to classify problems. The active measurements are performed using the PlanetLab infrastructure unless otherwise noted. We now present an overview of each of these measurements, and then elaborate on how the system uses these measurements to monitor and classify reachability problems.

Pingable address discovery: Pingable address discovery supplies the set of destinations for monitoring to the

active ping monitoring system. It discovers these destinations by probing the .1 in every /24 prefix present in a BGP snapshot obtained from RouteViews [31] and retaining those that respond.

Active ping monitors (details in §3.3.1): *Hubble* issues ping probes from vantage points around the world to the pingable addresses. The system in aggregate probes each address every two minutes. When a vantage point discovers a previously responsive address failing to respond, it reports the prefix as a candidate potentially experiencing more widespread reachability problems, resulting in triggered traceroutes to the prefix.

Passive BGP monitor (§3.3.2): The system observes BGP updates from multiple locations in quarter-hourly batches to maintain current AS-level routing information. This approach allows continuous monitoring in near real-time of the entire Internet from diverse viewpoints, while providing scalability by gathering information without issuing active probes. Supplementing its ping monitoring, *Hubble* analyzes the BGP updates and identifies as targets for triggered traceroutes those prefixes undergoing BGP changes, as they may experience related reachability problems. BGP feeds also allow *Hubble* to monitor prefixes in which no pingable addresses were found and hence are not monitored by the active ping monitors.

Triggered traceroute probes (§3.4.1): Every 15 minutes, *Hubble* issues active traceroute probes from distributed vantage points to targets selected as potentially undergoing problems. The system selects three classes of targets: (1) previously reachable addresses that become unreachable, as identified by the active ping monitors; (2) prefixes undergoing recent BGP changes, as identified by the passive BGP monitor; and (3) prefixes found to be experiencing ongoing reachability problems in the previous set of triggered probes.

Daily topology mapping (§3.5.1): If *Hubble* only launched traceroutes when it suspected a problem, these triggered probes would not generally give the system a view of what routes looked like before problems started. To supplement the triggered traceroutes, the system also maps the structure of the entire Internet topology using daily traceroutes, supplemented with probes to identify which network interfaces are collocated at the same router. This provides a set of baseline routes and a structured topology to map network interfaces to routers and ASes.

Router liveness monitors: Each vantage point monitors the routers on its paths from the previous day by issuing quarter-hourly pings to them. When a prefix becomes unreachable, *Hubble* uses these pings to discern whether the routers on the old path are still reachable, helping to classify what happened.

Spoofed probes (§3.5.4): Internet routes are often asymmetric, differing in the forward and reverse direction [23]. A failed traceroute signals that at least one direction is not functioning, but leaves it difficult or impossible to infer which. We employ spoofed probes, in which one monitor sets the source of packets to the IP of another monitor while probing a problem prefix. This technique aids in classification by isolating many problems to either the forward or reverse path.

3.3 Identifying Targets for Analysis

Selective targeting allows the system to monitor the entire Internet with limited active probing by identifying as targets for analysis only prefixes suspected to be experiencing problems. *Hubble* uses a hybrid approach, combining active ping monitoring with passive BGP monitoring. If *Hubble* used only passive BGP monitoring, it would miss any reachability event that did not correlate with BGP updates; as we present later in Section 4, BGP is not a good predictor of most problems, but allows *Hubble* to identify more problems than ping monitoring alone. We now present more details on how the two monitoring subsystems work.

3.3.1 Active Ping Monitoring

To meet our goal of a system with global scale, *Hubble* employs active monitoring of the reachability of prefixes. *Hubble* uses traceroute probes to perform its classification of reachability problems. However, it is not feasible to constantly traceroute every prefix in order to detect all problems. On heavily loaded PlanetLab machines, it would take any given vantage point hours to issue a single traceroute to every prefix, and so problems that were only visible from a few vantage points might not be detected in a timely manner or at all.

Hubble's active ping monitoring subsystem achieves the coverage and data-plane focus of active probing, while substantially reducing the measurement overhead versus a heavy-weight approach using pervasive traceroutes. If a monitor finds that a destination has become unresponsive, it reports the destination as a target for triggered traceroutes.

We design the ping monitors to discover as many reachability problems as possible, while reducing the number of spurious traceroutes sent to prefixes that are in fact reachable or are experiencing only transient problems. When a particular vantage point finds an address to be unresponsive, it reprobates 2 minutes later. If the address does not respond 6 times in a row, the vantage point identifies it as a target for reachability analysis, triggering distributed traceroutes to the prefix. We found that delaying the reprobates for a few minutes eliminates most transient problems, and we conducted a simple measurement study that found that the chance of a response on a

7th probe after none on the first 6 is less than 0.2%. 30 traceroutes to a destination entail around 500 total probe packets, so a 0.2% chance of a response to further pings means that it requires fewer packets to trigger traceroutes immediately, justifying launching them from distributed vantage points to investigate the problem.

A central controller periodically coordinates the ping monitors, such that (including reprobings) at least one but no more than six should probe each destination within any two minute period. Once a day, the system examines performance logs, replacing monitors that frequently fall behind or report improbably many or few unresponsive destinations. *Hubble* thus regularly monitors every destination, discovering problems quickly when they occur, without having the probing be invasive.

3.3.2 Passive BGP Monitoring

Hubble uses BGP information published by RouteViews [31] to continually monitor nearly real-time BGP routing updates from more than 40 sources. *Hubble* maintains a BGP snapshot at every point in time by incorporating new updates to its current view. Furthermore, it maintains historical BGP information for use in problem detection and analysis.

Hubble uses BGP updates for a prefix as an indicator of potential reachability problems for that prefix. In some cases, reachability problems trigger updates, as the change in a prefix from being reachable to unreachable causes BGP to explore other paths through the network. In other cases where the reachability problem is due to a misconfigured router advertising an incorrect BGP path, BGP updates could precede a reachability problem. We therefore use BGP updates to generate targets for active probes. Specifically, we select those prefixes for which the BGP AS path has changed at multiple vantage points or been withdrawn.

3.4 Real-time Reachability Analysis

Given a list of targets identified by the ping and BGP monitors, *Hubble* triggers traceroutes and integrates information from up-to-date BGP tables to assess the reachability of the target prefixes.

3.4.1 Triggered traceroutes

The daily traceroutes are of limited utility in identifying reachability problems that last only a few hours. The alternative of constantly performing traceroutes to every prefix is both inefficient and impractical. Nor do we want to sacrifice the level of detail exposed by traceroutes regarding actual routing behavior in the Internet, especially since such detail can then be used to localize the problem. *Hubble* strikes a balance by using triggered traceroutes to target prefixes identified by either the passive BGP monitor or the active ping monitors, plus prefixes known to be experiencing ongoing reacha-

bility problems. So, as long as a routing problem visible from our PlanetLab vantage points persists, *Hubble* will continually reprobe the destination prefix to monitor its reachability status.

Every 15 minutes, *Hubble* triggers traceroutes to the destinations on the target list from 30 PlanetLab nodes distributed across the globe. We limit these measurements to only a subset of PlanetLab nodes. Traceroutes from over 200 PlanetLab hosts within a short time span might be interpreted by the target end-hosts as denial of service (DoS) attacks. In the future, we plan to investigate supplementing the PlanetLab traceroutes with measurements from public traceroute servers; for example, when AS *X* suddenly appears in AS paths announced for a given prefix, *Hubble* could issue traceroutes to that prefix from traceroute servers that *X* makes available.

3.4.2 Analyzing Traceroutes to Identify Problems

In this section, we describe how *Hubble* identifies that a prefix is experiencing reachability problems. The analysis uses the triggered traceroutes, combined with *Hubble*'s passive routing view as obtained from RouteViews.

Since *Hubble* chooses as probe targets a single .1 in each of the suspect prefixes, we cannot be assured of a traceroute reaching all the way to the end-host, even in the absence of reachability problems. In some cases, a host with the chosen .1 address may not even exist, may be offline, or may stop responding to ICMP probes. Hence, we take a conservative stance on when to flag a prefix as being unreachable. We consider the origin AS for this prefix at the time when we issue the traceroute and flag the traceroute as having reachability problems if it does not terminate in the origin AS. We do this by mapping the last hop seen in the traceroute to its prefix and then to the AS originating that prefix in the BGP snapshot. Rarely, a prefix may have multiple origins [36], in which case we consider the set of ASes. We also consider the origin ASes of aliases of the last hops; if the last network interface seen on the traceroute has an alias (i.e., another IP address that belongs to the same router), and if the alias is within the address space of the origin AS, then we consider the destination reachable.

Note that, because we define reachability based on the origin AS for a prefix in routing tables, *Hubble* ignores prefixes that are completely withdrawn; these prefixes are easily classified as unreachable just by observing BGP messages. Further, note that our reachability check matches against any of the BGP paths for the prefix, rather than the particular path visible from the source of the traceroute. This is because *Hubble* issues traceroutes from PlanetLab nodes, while it gets its BGP data from RouteViews' vantage points, and the two sets are disjoint.

Traceroutes may occasionally not reach the destina-

tion for reasons that have little to do with the overall reachability of the target prefix, such as short-lived congestion on a single path or problems near the source. As described in Section 2.2, we flag a prefix as experiencing reachability problems worth monitoring only if less than 90% of the triggered probes to it reach the origin AS.

3.5 Topological Classification of Problems

As described thus far, *Hubble* only identifies prefixes experiencing problems, without pointing to the locations of the problems. To address a problem, an operator would like to know (a) the AS in which the problem occurs, to know who to contact; (b) which of the AS's routers could be causing problems; and (c) whether the issue is with paths to or from the problem prefix. Sections 3.5.2 and 3.5.3 describe how *Hubble*'s infrastructure enables classification of problems in a way that begins to address (a) and (b). In Section 3.5.4, we describe how we use spoofed probes to deal with (c). First, we describe some of the measurements that aid in classification.

3.5.1 Daily Topology Mapper

To aid in its classification, *Hubble* performs traceroutes daily to the destinations identified by the pingable address discovery. More than 200 PlanetLab sites performed traceroutes to each of these destinations once a day for the past year, and we plan to keep these daily traceroutes running continuously for the foreseeable future. These daily traceroutes enable *Hubble* to maintain a set of fairly recent base paths from each host to each destination. These base paths provide a comparison when a problem occurs; if we probed a prefix only when it was having problems, we might not know how the working paths looked before the problem. When a problem develops, *Hubble* can ping routers along the old path to determine if they remain reachable.

Additionally, we use information from the daily traceroutes to identify router interfaces, which are IP addresses belonging to different interfaces on the same router. To collect this information, we identify a list of about 2 million interfaces from our daily traceroutes and from pings to all our pingable addresses with the record route IP option enabled. The traceroutes return incoming interfaces on routers on the paths from our vantage points to the destinations, whereas the record route option-enabled pings return outgoing interfaces on routers. We identify alias candidates among these using the Mercator technique [11] (for which we probe all the interfaces using UDP probes) and the heuristic that interfaces on either end of a link are commonly in the same /30 prefix. We then probe each pair of alias candidates in succession with UDP and ICMP probes. We classify a pair of interfaces as aliases only if they return similar IP-IDs and similar TTLs [26]. *Hubble* uses this alias informa-

tion in analyzing prefix reachability, as discussed in Section 3.4.2.

3.5.2 Hubble's Approach to Classification

Without access to complete topologies, direct BGP feeds from every AS, real-time status of router queues, and router configurations, it is often impossible to pinpoint the exact reason a given probe fails to reach its destination. Our approach with *Hubble* is to identify network entities (ASes, routers, links, or interfaces) that seem to explain the failure of a substantial number of probes to a given prefix in a round of probes. We define a reachability problem as when traceroutes from less than 90% of vantage points reach the origin AS for the prefix, so we say that an entity explains a substantial number of failed probes if it accounts for 10% or more of the vantage points. We do not require it to explain all failed probes in the set, and we may classify a problem prefix in multiple ways at once. Multiple classifications could indicate multiple simultaneous problems, multiple problems with a single root cause, or evolving problems as operators or automatic processes react or problems cascade.

Hubble's simple classification scheme relies on grouping failed probes based on the last observable hop, the expected next hop, and the ASes of each of these hops. *Hubble* infers the next hop from its historical record of working paths. We emphasize that the approach does not necessarily pinpoint the exact entity responsible; the problem could, for instance, occur on the handoff from the entity or on the return path. We address this second issue in Section 3.5.4. Our classification goals are modest—we do not currently assign blame, but simply illustrate where the traceroutes terminate. We believe this information provides a useful starting point for operators determining the exact cause of problems. In the future, we plan to expand *Hubble*'s classification abilities, as well as provide verification based on known outages and communications with operators.

3.5.3 Classes

Hubble currently automatically assigns, in real-time, reachability problems into 9 classes, when appropriate. These classes represent different topological patterns of which traceroutes reach and which fail to reach, and they were based on preliminary hand-analysis of observed problems and chosen because they appeared to cover a substantial number of cases. Note that we infer origin and provider ASes based on active routes for the prefix in our BGP tables during the time period of the probes. In the following discussion of the classes, the destination is in prefix *P*, originated by AS *O*. We say that a probe reaches an AS if the longest matching prefix of an interface observed in the traceroute is originated by the AS,

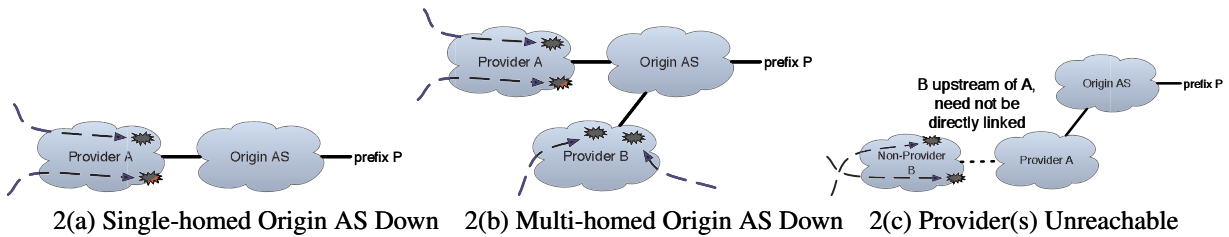


Figure 2: Classes of complete unreachability, meaning all traceroutes fail to reach the origin AS. In (a) the origin AS has a single announced provider for the prefix, whereas in (b) it has at least 2. In both cases, some traceroutes have a hop within the provider(s). In (c) all traceroutes terminate before the provider(s).

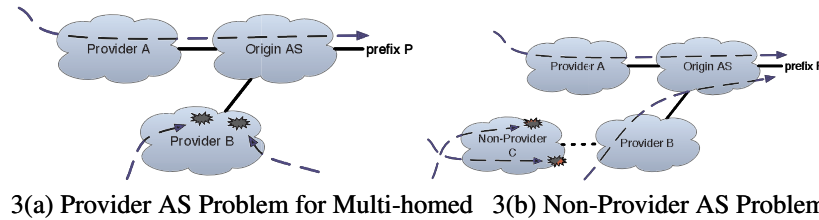


Figure 3: Classes of partial reachability in which all traceroutes reaching a particular AS fail, whereas some paths work through other ASes. In (a) the AS is a provider for the origin AS, whereas in (b) it is not.

or if one of the interfaces observed in the traceroute is an alias for an address originated by the AS.

The first three classes are cases of complete unreachability, with no traceroutes reaching even the origin AS for the prefix. They are illustrated in Figure 2.

Single-homed Origin AS Down: In this classification, none of the probes reach O , but some of the probes reach O 's provider, A . Further, active AS paths for the prefix contain A as the only upstream provider for O .

Multi-homed Origin AS Down: This classification is the same as the previous, except that O has more than one provider in active BGP paths for P .

Provider(s) Unreachable: In this classification, none of the traceroutes reach the provider(s) of O , and a substantial number terminate in an AS further upstream.

Whereas in the previous classes no probes reach the prefix, the next five cover cases when some do. In the next two, all traceroutes reaching a particular AS terminate there. They are illustrated in Figure 3(a) and (b).

Provider AS Problem for Multi-homed: In this classification, all probes that reach a particular provider B of origin AS O fail to reach O , but some reach P through a different provider A . This classification is particularly interesting because ASes generally multi-home to gain resilience against failure, and an occurrence of this class may indicate a problem with multi-homed failover.

Non-Provider AS Problem: In this classification, all probes that reach some AS C fail, where C is not a direct provider of O but rather is somewhere further upstream. Some probes that do not traverse C successfully reach P .

The previous five classes represent cases in which all

probes that reach some AS fail to reach the prefix. In the next two classes, all probes that reach a particular router R fail to reach P and have R as their last hop, but some probes through R 's AS successfully reach P along other paths. These classes are illustrated in Figure 4(a) and (b).

Router Problem on Known Path: In this classification, R appeared on the last successful traceroute to P from some vantage point, and so the historical traceroute suggests what the next hop should be after R .

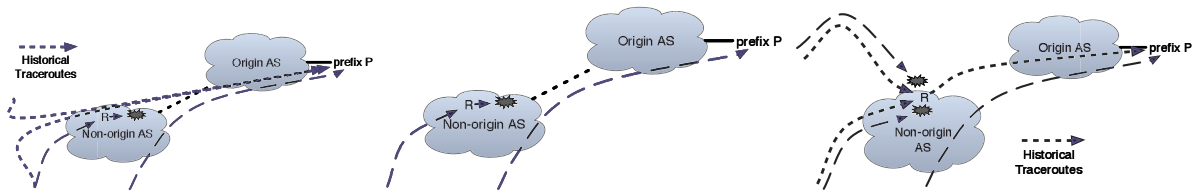
Router Problem on New Path: This classification is similar to the last, the difference being that R did not appear on the last successful traceroute to P from any vantage point. So, the problem may be due to a path change or a failure on the old path.

Next Hop Problem on Known Paths: In this classification, illustrated in Figure 4(c), no last hop router or AS explains a substantial number of failed probes. However, based on the last successful paths from some vantage points, the probes that should have converged on a particular next hop all terminated right before it.

We defined the previous 5 classes for cases of partial reachability in which some of the probes reach the prefix. All 5 have analogous versions in which some probes reach the origin AS, but none reach the prefix. We consider these less interesting, as the prefix is either down or a problem exists within the origin. So we classify them together as **Prefix Unreachable**.

3.5.4 Differentiating Failures using Spoofed Probes

The classes described above help guide an operator in searching for the causes of outages, but still leave open various explanations. Consider Figure 3 (a), with a probe



4(a) Router Problem on Known Path 4(b) Router Problem on New Path 4(c) Next Hop Problem on Known Paths

Figure 4: *Classes of partial reachability in which some paths through an AS work and others do not. Lines with shorter dashes indicate the last successful traceroute from some vantage point, whereas longer dashes indicate traceroutes from the current round of probes. In (a) and (b), all traceroutes reaching a particular router fail. In (a) the router is on known paths, in (b) on a new path. In (c) paths that previously converged at a router and reached the prefix now stop just before that router.*

from monitor a reaching through provider A , and probes from monitors b_1 and b_2 terminating with last hops in B . One might assume that the problem is between B and the origin, but it could also be a problem on the return paths to b_1 and b_2 . With just the forward path information supplied by traceroutes, these cases are indistinguishable. We employ spoofed probes to differentiate the cases and provide much more specific information about the failure. Note that we only ever spoof packets using the source address of one of our other vantage points.

To determine why b_1 cannot reach P , monitor a sends probes to P with the source set as b_1 . These probes reach P along a 's forward path. If the responses to these probes reach b_1 , then we know that the reverse path from P to b_1 works, and we determine that the failure is on b_1 's forward path. Otherwise, b_1 sends probes to P with the source set as a . If b_1 's forward path works, then the responses to these probes should reach a , and we determine that the failure is on the reverse path back to b_1 . A central controller coordinates the spoofing, assigning, for instance, a to probe P spoofing as b_1 , then fetching the results from b_1 for analysis. Because we only draw conclusions from the receipt of spoofed probes, not their absence, we do not draw false conclusions if b_1 does not receive the probe for other reasons or if the controller is unable to fetch results.

Currently, the PlanetLab kernel does not allow spoofed probes to be sent, and so spoofing is not fully integrated into *Hubble*. In Section 5.3, we provide preliminary results using a parallel deployment on RON [1], plus discuss possibilities for better future integration.

3.6 Problem Granularity and Probe Rate

In an early version of *Hubble*, with target selection based only on RouteViews updates, which are published in quarter-hourly batches, we discovered a surprising number of long-lasting problems and decided that this coarseness was still interesting. We then based our design on techniques that suffice for this granularity, with probing in quarter-hourly rounds. Besides providing abundant problems, we have other reasons for favoring long-lasting problems. First, short problems have been studied

and are expected during BGP convergence [18]. Second, short-lasting problems, by definition, already resolve quickly, so are less in need of a system to identify them to operators.

It is worth considering what it would take for a future version of *Hubble* to discover shorter problems. With PlanetLab having a few hundred sites, we have a greater than 80% chance of discovering any reachability problem after 15 sites have probed the prefix, as by our definition it must be unreachable from $> 10\%$ of sites. (Since probing order is random, we consider probes to be independent.) To reliably discover minute-long problems, then, would require probing each destination every 4 seconds, a rate sustainable with our current deployment. An order of magnitude faster than that would likely require retooling the system, given the limited number of PlanetLab sites and the high load on them. However, archives of the PlanetLab Support mailing list reveal that experiments pinging destinations 4 times per second generate multiple complaints. It is unclear to us what probe rates are tolerable to ISPs.

Hubble could maintain the level of traceroutes necessary for minute-long problem discovery by increasing the number of parallel probing threads on each site and the number of sites used for triggered traceroutes (such that different sets of vantage points probe different problems). Further, a non-dedicated machine with a single Intel 3.06 GHz Xeon processor and 3 GB RAM can execute a round of reachability analysis (as MySQL scripts) in about a minute. We suspect we could significantly reduce it to keep pace with a faster probe rate, as we have not sought to optimize it given that this running time suffices for our current needs. Further, although cross-prefix correlation is an interesting future direction, current analysis is per-prefix, so the load could easily be split across multiple database servers.

4 Evaluation

Hubble is now a continuously running, automated system that we plan to keep up, minus maintenance and upgrades. We start by giving an example of one of the problems *Hubble* found. On October 8, 2007, at

5:09 a.m. PST, one of *Hubble*'s ping monitors found that 128.9.112.1 was no longer responsive. At 5:13, *Hubble* triggered traceroutes from around the world to that destination, part of 128.9.0.0/16, originated by USC (AS4). 4 vantage points were unable to reach the origin AS, whereas the others reached the destination. All of the failed probes stopped at one of two routers in Cox Communications (AS22773), one of USC's providers, whereas the successful probes traversed other providers. In parallel, 6 of 13 RON vantage points were unable to reach the destination, with traceroutes ending in Cox, while the other 7 RON nodes successfully pinged the destination. *Hubble* launched pings from some of those 7 nodes, spoofed to appear to be coming from the other 6, and all 6 nodes received responses from 128.9.112.1. This result revealed that the problems were all on forward paths to the destination, and *Hubble* determined that Cox was not successfully forwarding packets to the destination. It continued to track the problem until all probes launched at 7:13 successfully reached the destination, resolving the problem after 2 hours. A snapshot of the problems *Hubble* is currently monitoring can be found at <http://hubble.cs.washington.edu>.

In this section, we evaluate many of our design decisions to assess *Hubble*'s efficacy. In Section 5 we present results of a measurement study conducted using it.

How much of the Internet does *Hubble* monitor? *Hubble* selects targets from BGP updates for the entire routing table available from RouteViews. Its active ping monitoring includes more than 110,000 prefixes discovered to have pingable addresses, distributed over 92% of the edge ASes, i.e., ASes that do not provide routing transit in any AS paths seen in RouteViews BGP tables. These target prefixes include 85% of the edge prefixes in the Internet and account for 89% of the edge prefix address space, where we classify a prefix as non-edge if an address from it appears in any of our traceroutes to another prefix. Previous systems that used active probes to assess reachability managed to monitor only half as many ASes over 3 months and only when clients from those ASes accessed the system [35], whereas *Hubble* probes each of its target prefixes every 2 minutes.

We next gauge whether *Hubble* is likely to discover problems Internet users confront. To do so, we collected a sample of BitTorrent users by crawling popular sites that aggregate BitTorrent metadata and selecting 18,370 target swarms. For a month starting December 20, 2007, we repeatedly requested membership information from the swarms. We observed 14,380,622 distinct IPs, representing more than 200 of the nearly 250 DNS country codes. We are interested in whether the routing infrastructure provides connectivity, and so *Hubble* monitors routers rather than end-hosts, which are more likely to go offline (and do not affect others' reachability when

they do) and often are in prefixes that do not respond to pings. Further, a router generally uses the same path to all prefixes originated by a given AS [19]. Therefore, we assess whether these representative end-users gathered from BitTorrent share origin ASes with routers monitored by *Hubble*. We find that 99% of them belong to ASes containing prefixes monitored by *Hubble*.

How effective are *Hubble*'s target selection strategies?

To reduce measurement traffic overhead while still finding the events that occur, *Hubble* uses passive BGP monitoring and active ping monitoring to select targets likely to be experiencing reachability problems. Reachability analysis like *Hubble*'s relies on router-level data from traceroutes (see Sections 3.4 and 3.5). So we compare the ability of *Hubble*'s selective targeting to discover problems with an approach using pervasive tracerouting, in which the 30 vantage points each probe all monitored prefixes every 15 minutes without any target selection. We measured the total probe traffic sent by *Hubble*, including pings and traceroutes, and found that it is 5.5% of that required by the pervasive technique.

Given its much reduced measurement load, we next assess how effectively *Hubble*'s target selection strategies discover events compared to pervasive traceroutes. For this evaluation, we issued traceroutes every 15 minutes for ten days beginning August 25, 2007, from 30 PlanetLab vantage points to 1500 prefixes, and we compare the reachability problems discovered in these traceroutes with those discovered to the same set of prefixes by *Hubble*'s BGP- and ping-based target selection. We use the quarter-hourly traceroutes as "ground truth" reachability information. We only consider events that both begin and end within the experiment and only consider events that persist for at least one additional round of probing after they start. There were 1100 such reachability events, covering 333 of the prefixes, with the longest lasting almost 4 days. 236 of the events involved complete unreachability, and 874 were partial. Here and in later sections, we classify a reachability event as being complete if, at any point during the event, none of the traceroute vantage points is able to reach it. Otherwise, the event is partial.

Figure 5 shows the fraction of the events also uncovered by *Hubble*'s target selection strategies, both individually and combined. Individually, active ping monitoring uncovered 881 of the problems (79%), and passive BGP monitoring uncovered 420 (38%); combined, they discovered 939 (85%). For events lasting over an hour, the combined coverage increases to 95%. The average length of an event discovered by ping monitoring is 2.9 hours, whereas the average length of an event discovered by BGP monitoring and not by ping monitoring is only 0.8 hours.

This experiment yields a number of interesting con-

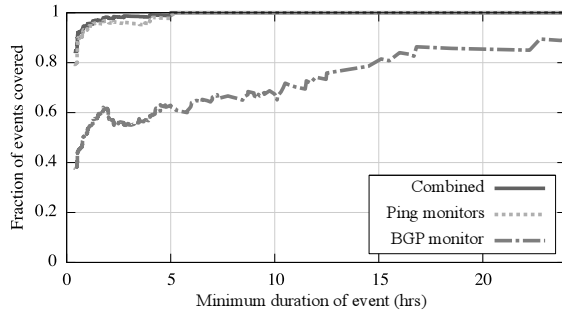


Figure 5: For reachability events discovered in 10 days of quarter-hourly probes, fraction of events also discovered by Hubble’s target selection. While BGP alone proved ineffectual, Hubble’s techniques combine to be nearly as effective as a heavy-weight approach with the same time granularity.

clusions. First, BGP monitoring is not sufficient. We were surprised at how low BGP-based coverage was; in fact, we had originally intended to only do BGP based monitoring, until we discovered that it uncovered too few events. Second, BGP monitoring provides an important supplement to active monitoring, particularly with short events. Because we strive to limit the rate at which we probe destinations, an inherent tradeoff exists between the number of monitors (more yielding a broader viewpoint) and the rate at which a single monitor can progress through the list of ping targets. In our current implementation, we use approximately 100 monitor sites, and it takes a monitor over 3 hours to progress through the list. Therefore, short reachability problems visible from only a few vantages may not be discovered by ping monitors. BGP monitoring often helps in these cases. Third, *Hubble*’s overall coverage is excellent, meaning it discovers almost all of the problems that a pervasive probing technique would discover, while issuing many fewer probes.

How quickly after they start does Hubble identify problems? Besides uncovering a high percentage of all reachability events, we desire *Hubble* to identify the events in a timely fashion, and we find that it does very well at this. For the same reachability events as in Figure 5, Figure 6 shows the delay between when the event starts in the quarter-hourly probes and when the prefix is identified as a target by *Hubble*’s target selection. Because of the regular nature of the quarter-hourly probes, we know the actual starting time of the event to within that granularity. However, it is possible that *Hubble*’s monitoring identifies problems before the “continuous” traceroutes; in these cases, for ease of readability, we give the delay as 0. We additionally plot events lasting longer than an hour separately to avoid the concern that the large number of events shorter than that might distort *Hubble*’s performance. The ideal plot in the graph would be a vertical line at 0; *Hubble* achieves that for 73% of the events it identifies, discovering them at least

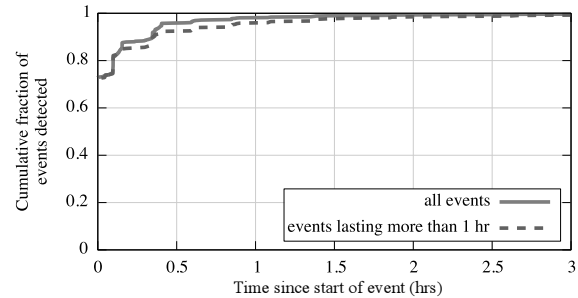


Figure 6: For reachability events in 10 days of quarter-hourly probes, time from start of event until Hubble identifies its prefix as a target. Events over an hour are also given separately. Fractions are out of only the events eventually identified, 85% overall and 95% of those longer than an hour. Hubble identifies 73% of events immediately.

as early as quarter-hourly probes. Of the events lasting over an hour, *Hubble* discovers 96% of them within an hour of the event’s start. So *Hubble*’s light-weight probing approach still allows it to discover events in a timely fashion, and we can generally trust the duration it gives for the length of an event.

Does Hubble discover those events that affect sites where it does not have a vantage point? One limitation of the evaluation so far is that the 30 PlanetLab sites used to issue the quarter-hourly traceroutes are also used as part of the ping monitoring. We would like *Hubble* to identify most of the reachability problems that any vantage points would experience, not just those experienced by its chosen vantages. To partially gauge its ability to do this, we assess the quality of its coverage when we exclude the traceroute vantage points from the set of ping monitors. This exclusion leaves *Hubble* with only about $\frac{2}{3}$ of its normal number of monitors, and the excluded vantage points include 4 countries not represented in the remaining monitors. Yet our system still discovers 77% of the 1110 reachability events (as compared to 85% with all monitors). If we instead exclude an equal number of vantage points chosen randomly from those not issuing traceroutes, we see 80% coverage (median over 3 trials). We acknowledge that known diversity issues with PlanetLab somewhat limit this experiment.

To assess this limitation, we evaluate the diversity of paths seen from PlanetLab compared to BGP paths from the RIPE Routing Information Service [30], which is similar to RouteViews but with many more AS peers (447 total). The research community believes Internet routes are generally valley-free, with “uphill” and “downhill” segments. For each RIPE path, we consider only the segment from the highest degree AS to the prefix (the downhill portion); we truncate in this manner because we found more than 90% of failed traceroutes terminate within two AS hops of the destination’s origin

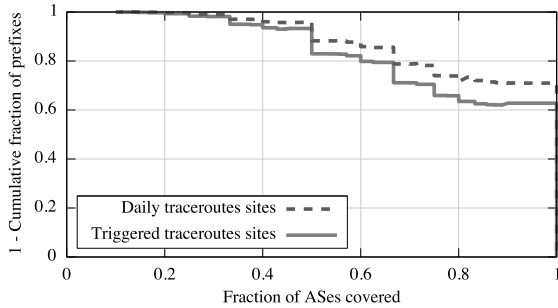


Figure 7: Fraction of ASes on BGP paths from RIPE RIS route collectors that also appear on traceroutes from Hubble’s daily and triggered traceroute vantage points. Using only 35 sites for triggered traceroutes, Hubble observes most of the ASes visible from the 218 PlanetLab sites and the 447 RIPE peers.

AS, and the relatively small number of PlanetLab sites limits the source-AS diversity on the uphill portion of paths. For each prefix monitored by *Hubble*, we consider the set of all ASes that appear on the truncated RIPE paths for that prefix. We also calculate the set of ASes that appear on one day’s worth of *Hubble*’s daily traceroutes to each prefix. Figure 7 shows the fraction of ASes on RIPE paths also seen in daily traceroutes. Even with just the 35 sites used for *Hubble*’s triggered traceroutes, for 90% of prefixes, the probes include at least half of the ASes seen in BGP paths. For 70% of prefixes, the traceroutes include at least 70% of ASes, and they include all ASes for more than 60% of prefixes. These results suggest that PlanetLab achieves reasonable visibility into AS paths even with a small number of vantage points, so *Hubble* likely detects many of the AS problems that occur on the downhill portions of paths to its monitored prefixes. Further, limiting triggered probe traffic during problems to a small number of vantage points does not drastically reduce the system’s coverage. While the system currently selects a single set of vantage points, seeking only to maximize the number of source countries without considering AS-path redundancy, we could easily modify it to use daily traceroutes to choose triggered traceroute sites on a per-prefix basis to maximize path diversity.

We have future plans to extend *Hubble*’s view which we mention briefly in Section 5.3; further analysis of RIPE BGP paths could suggest where our coverage is most lacking. Even now, three facts allow *Hubble* to discover many of the problems experienced by sites outside of its control. First, passive BGP monitoring gives *Hubble* a view into ASes outside of its control. Second, as noted in Section 2.2, when a problem exists, it is quite likely that many vantage points experience it. Third, as we will see in Section 5.2, many problems occur near the destinations, by which point paths from many diverse vantage points are likely to have converged.

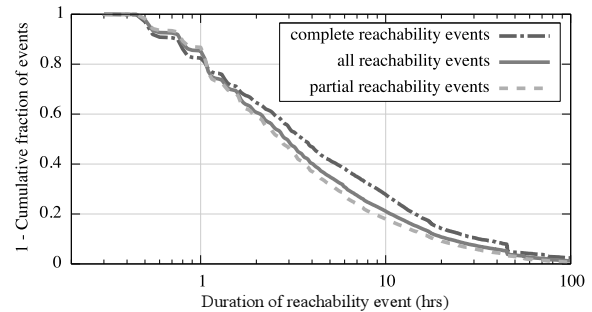


Figure 8: CCDF of duration of reachability events.

5 Characteristics of Reachability Problems on the Internet

After demonstrating the effectiveness of *Hubble* in achieving our goals, we now present the results of a measurement study using *Hubble* to detect and measure reachability problems on the Internet for 3 weeks starting September 17, 2007. *Hubble* issued traceroutes from 35 PlanetLab sites across 15 countries (though only 30 at a time) and deployed ping monitors at 104 sites across 24 countries. In Section 2.2, we defined a reachability problem to be when a prefix is reachable from less than 90% of probing sites, and a reachability event is the period starting when we first identify that a prefix is experiencing reachability problems and concluding when its reachability increases to 90% or higher. We consider only events that began and ended during the study and persisted through at least one additional round of probing after being detected.

5.1 Prevalence and Duration

Hubble identified 31,692 reachability events, involving 10,224 distinct prefixes. 21,488 were cases of partial reachability, including 6,202 prefixes. 4,785 prefixes experienced periods of complete unreachability. *Hubble* detected an additional 19,150 events that either were transient or were still ongoing at the end of the study, involving an additional 6,851 prefixes. Of the prefixes that had problems, 58% experienced only a single reachability event, but 25% experienced more than 2 and 193 experienced at least 20.

Figure 8 shows the duration of reachability events. More than 60% lasted over 2 hours. From Section 4, we know the system has excellent coverage of events this long, but may miss some shorter ones. Still, this represents over 19,000 events longer than 2 hours, and 2,940 of the events lasted at least a day. Cases of partial reachability tend to resolve faster, with a median duration of 2.75 hours, $\frac{3}{4}$ of an hour shorter than for cases of complete unreachability. Even so, in 1,675 instances a prefix experienced partial reachability for over a day. We find this to be an astounding violation of global reachability.

5.2 Topological Characteristics

We conducted a study of *Hubble*'s reachability problem classification, applied to the triggered traceroutes issued in the first week of February, 2008. If a set of 30 probes indicates a prefix is experiencing reachability problems, *Hubble* attempts in real-time to automatically match the problem to one of the classes presented above. We first present a few case studies, then give quantitative results of *Hubble*'s classification. We intend the case studies to serve as examples of problems *Hubble* detects, but do not mean them to be exhaustive. *Hubble* classified these problems automatically, but we followed up by hand to get details such as the ASes involved.

Example of complete unreachability: For a prefix originated by an AS in Zimbabwe, probes to routers along previously successful paths to the prefix showed that the link to its primary provider seemed to have disappeared, and traffic was being routed through a backup provider. However, all probes terminated in this backup provider, either due to a misconfiguration in the secondary provider or due to the origin AS being down. In subsequent rounds of probing, packets started getting through to the destination only after the link to the primary provider came up again. This type of problem cannot be detected without active measurements, as the backup exported a valid AS path.

Example of partial reachability, AS problem: *Hubble* found that all probes to a particular prefix in Hong Kong that went through *FLAG Telecom* were dropped, whereas those that used other transit ASes reached the destination AS, *Hutchinson*. Of the 30 traceroutes to this destination, 11 went through *FLAG* and failed to reach the destination. This observation strongly suggests problems with the *FLAG-Hutchinson* connection.

Example of partial reachability, router problem: We saw an example of this scenario for an AS in Vietnam. Probes from 15 of our vantage points passed through the *Level 3* network, with some of the probes being dropped in *Level 3* while others reached the destination. Comparing the failed probes with earlier ones in which all 15 probes through *Level 3* were successful, we observed that the internal route within *Level 3* had changed. In the earlier successful traceroutes, packets reached a router 4.68.120.143 in the *Level 3* network and were forwarded to another router 213.244.165.238 (also in *Level 3*), and then to the destination AS. However, in the failed probes flagged as a reachability problem, packets reached router 4.68.120.143, which then sent them to another *Level 3* router 4.68.111.198, where the traceroutes terminated. This path change could have been due to load balancing or changes in IGP weights, because all the routers on the old path, including router 213.244.165.238, still

Class	Total %	Min %	Max%
Single-homed Origin AS Down	17	4	37
Multi-homed Origin AS Down	9	2	30
Provider(s) Unreachable	3	1	13
Provider AS Problem for Multi-homed	6	1	17
Non-Provider AS Problem	17	1	37
Router Problem on Known Path	7	1	40
Router Problem on New Path	21	1	40
Next Hop Problem on Known Paths	14	1	39
Prefix Unreachable	22	7	79

Table 1: *Percentage of problems in each class in one week of triggered traceroutes. Total column gives the percentage belonging to that class, out of the 375,775 total classified; recall that, as explained above, a problem can be classified in multiple ways. Min (Max) column gives the percentage of problems assigned to that class, out of all problems classified during the 15 minute window, for the 15 minute window with the lowest (highest) percentage for that class.*

responded to *Hubble*'s pings. This implies that either router 4.68.111.198 is misconfigured, or that the routing information is not consistent throughout the AS.

Quantitative classification results: *Hubble* classified 375,775 of the 457,960 sets of traceroutes (82%) that indicated reachability problems during the study. The other problems were not classifiable using *Hubble*'s technique of grouping failed probes by last AS, last hop, or inferred next hop, then flagging any such entity that explains a substantial number. In those cases, every such entity either did not explain enough probes or had other probes that reached the destination through it, perhaps because the problem resolved while we were probing or because a problem existed on the return path to some vantage points and not others.

Table 1 shows how many problems were assigned to each class. *Hubble* classified 91.95% of all cases of complete unreachability, yielding almost one-third of the classified problems; especially for small ASes originating a single prefix, these may be cases when the prefix has simply been taken offline for awhile. The cases of partial reachability are more interesting, as a working physical path exists. Suppose s_1 is unable to reach d , but s_2 can. If nothing else, a path exists in which s_1 tunnels traffic to *Hubble*'s central coordinator (running at the University of Washington), to which it must have access as it reported d as unreachable, and *Hubble* tunnels traffic to s_2 , which forwards it on to d . While phys-

ical failure— say, a bulldozer mistakenly cutting fiber— can cause complete unreachability, any case of partial reachability must be caused at least in part by either policy or misconfiguration. Policy-induced unreachability and misconfigurations might also help explain why BGP proved to be a poor predictor of reachability problems, as seen in Section 4.

We make two observations for the cases of partial reachability. First, we were surprised how often all traffic to a particular provider of a multi-homed AS failed when other providers worked. This result indicates that multi-homed failover may warrant further study and suggests that ASes may want to monitor their reachability through all their providers, perhaps using *Hubble*. Second, most of the router problems were on new paths; we plan further analysis of *Hubble* data to determine how often the routers on the old path were still available.

5.3 Classification Results Using Spoofed Probes

We conducted two studies on the RON testbed [1] to evaluate how effectively *Hubble*’s spoofed probes determine if a problem is due to issues with the forward path to or with the reverse path from the destination. Our studies used 13 RON nodes, 6 of which permitted spoofing of source addresses.

In the first study, we issued pings every half hour for a day to destinations in all the prefixes known by *Hubble* to be experiencing reachability problems at that time. We then discarded destinations that were either reachable from all 13 nodes or unreachable from all, as spoofed probes provide no utility in such cases. For every partially reachable destination d and for each RON node r which failed to reach d , we chose a node r' that could both reach d and send out spoofed probes. We had r' send a probe to d with the source address set to r . If r received d ’s response, it indicated a working reverse path back from d to r . We concluded that a problem on the forward path from r to d caused the unreachability. Similarly, in cases when a node r was able to send spoofed probes and unable to reach d , we had r send out probes to d with the source address set to that of a node r' from which d was reachable. If r' received d ’s response, it demonstrated a working forward path from r to d , and hence we concluded that the problem was on the reverse path from d back to r . We issued redundant probes to account for random losses.

How often do spoofed packets isolate the failed direction? We evaluated 25,286 instances in which one RON node failed to reach a destination that another node could reach; in 53% of these cases, spoofing allowed us to determine that the failure was on the forward path, and in 9% we determined the failure to be on the reverse path. These results were limited by the fact that we could only verify a working forward path from the 6 nodes capa-

Class	Forward	Reverse	Mix	Unknown	Total
All destinations with reachability problems					
All nodes	49%	0%	1%	50%	3605
Spoofing nodes	42%	16%	3%	39%	2172
Multi-homed dests. classified as having provider problems					
All nodes	84%	0%	0%	16%	18762
Spoofing nodes	81%	0%	0%	19%	10628

Table 2: *Out of cases in which at least 3 vantage points failed to reach the destination, the %’s in which our technique using spoofed packets determined that all problems were on the forward path, all on the reverse path, or a mix of both. Also gives the % for which our system could not make a determination.*

ble of spoofing. Looking only at the 11,355 failed paths from sources capable of spoofing, we found the problem to be on the forward path in 47% of cases and on the reverse path in 21%. The remaining 32% may have had failures both ways, or transient loss may have caught packets. Our 68% determination rate represents a five-fold improvement over previous techniques [35], which were able to determine forward path problems in 13% of cases but not reverse path failures. In an additional 15% of cases, their technique inferred the failure of an old forward path from observing a path change, but made no determination as to why the new path had failed.

The success of our technique at isolating the direction of failure suggests that, once we have an integrated *Hubble* deployment capable of spoofing from all vantage points, we will be able to classify problems with much more precision, providing operators with detailed information about most problems.

When multiple sites cannot reach a destination, how often do spoofed probes show all failed paths to be in the same direction? We then evaluated the same data to determine when all the reachability issues from RON nodes to a particular destination could either be blamed entirely on forward paths to the destination or on reverse paths back from the destination. In each half hour, we considered all targets to which at least one RON node had connectivity and at least three did not. We then determined, for each target, whether forward paths were responsible for all problems; whether reverse paths were; or whether each failed path could be pinned down to one direction or the other, but it varied across sources. We then repeated the experiment, but considered only sources capable of spoofing and only destinations unreachable from at least 3 of these sources. The top half of Table 2 presents the results. We determined the failing direction for all nodes in half of the cases, with nearly all of them isolated to the forward direction (note that the 1% difference accounts for cases when some of the spoofing nodes had reverse path failures while other

nodes had forward path ones). When considering just the spoofing nodes, we were able to explain all failures in 61% of cases. In 95% of those, the problems were isolated to either reverse or forward paths only, meaning that all nodes had paths to the destination or that the destination had paths to all nodes, respectively.

What is the nature of multi-homed provider problems? We conducted the second study to further determine how well spoofing can isolate problems. We used the same setup as before for two weeks starting October 8, 2007, but this time considered in each round only destinations that *Hubble* determined were experiencing provider AS problems for a multi-homed origin (see Figure 3 (a)). We chose this class of problems because operators we spoke with about our classification study from Section 5.2 wanted us to give them further information about what was causing the multi-homed provider problems we saw. In addition to the measurements from the first spoofing study, every RON node performed a traceroute to each destination, which we used to find those that terminated in the provider identified by *Hubble* as the endpoint for a substantial number of triggered traceroutes. We considered cases in which at least 3 paths from RON nodes terminated in the provider AS and determined in which cases we could isolate all failures. The bottom half of Table 2 gives the results. We determined the direction of all failures in more than $\frac{4}{5}$ of cases, and we were surprised to discover that all the problems were on the forward path. It seems that, in hundreds of instances a day, destinations across the Internet are reachable only from certain locations because one of their providers is not forwarding traffic to them.

What are the long term prospects for isolating the direction of failures? The above studies were limited to 13 RON nodes receiving spoofed probes, with 6 of them sending the probes. We have since developed the means to receive spoofed probes from RON nodes at all PlanetLab sites, allowing us to isolate forward path failures from the sites. Furthermore, PlanetLab support has discussed allowing spoofed probes from PlanetLab sites in future versions of the kernel. We have received no complaints about our probes, spoofed or otherwise, so they do not appear to be annoying operators. A major router vendor is talking to us about ways to provide better support for measurements.

The Internet's lack of source address authentication proved very useful in isolating the direction of failures. A more secure Internet design might have allowed authenticated non-source "reply-to" addresses. Even without this and with some ISPs filtering spoofed traffic from their end-users, we expect future versions of *Hubble* to provide better isolation in two ways. First, we can replace spoofed probes with probes sent out from traceroute servers hosted at ASes behind problems, similar

to [2]. Second, we plan to deploy a measurement platform in various end-user applications which we expect will give us much wider coverage than any current deployment, allowing us to issue probes to *Hubble* vantage points from end-hosts in prefixes experiencing problems.

5.4 Summary

We found the extent of reachability problems to be much greater than we originally expected, with *Hubble* identifying reachability problems in around 10% of the prefixes it was actively monitoring and some of the problems lasting over a day.

The majority of reachability problems observed by *Hubble* fit into simple topological classes. Most of these were cases of partial reachability, in which a tunneling approach could utilize *Hubble* data to increase the number of vantage points able to reach the destination. Most surprisingly, we discovered many cases in which an origin AS was unreachable through one of its providers but not others, suggesting that multi-homing does not always provide the resilience to failure that it should.

6 Related Work

Most related work can be classified into three categories: passive monitoring at a global scale, active monitoring on a limited scale, and intra-domain monitoring using proprietary or specialized information and tools.

Passive BGP Monitoring: Numerous studies have modeled and analyzed BGP behavior. For instance, Labovitz et al. [18] found that Internet routers may take tens of minutes to converge after a failure, and that end-to-end disconnectivity accompanies this delayed convergence. In fact, multi-homed failover averaged three minutes. Mahajan et al. [21] showed that router misconfigurations could be detected with BGP feeds. Caesar et al. [3] proposed techniques to analyze routing changes and infer why they happen. Feldman et al. [8] were able to correlate updates across time, across vantage points, and across prefixes; they can pinpoint the likely cause of a BGP update to one or two ASes. Wang [33] examined how the interactions between routing policies, iBGP, and BGP timers lead to degraded end-to-end performance. BGP beacons [22] benefited this work and other studies. Together, these studies developed techniques to reverse-engineer BGP behavior, visible through feeds, to identify network anomalies. However, there are limits to such passive monitoring approaches. Though it is possible to infer reachability problems by passive monitoring [17], often times the presence of a BGP path does not preclude reachability problems and performance bottlenecks. Further, BGP data is at a coarse, AS-level granularity, limiting diagnosis.

Active Probing: Other studies used active probes to discover reachability problems. Paxson was the first to

demonstrate the frequent occurrence of reachability issues [23]. Feamster et al. [6] correlated end-to-end performance problems with routing updates. These and other studies [1, 32, 5, 9] are designed for small deployments that probe only between pairs of nodes, allowing detailed analysis but limited coverage. Pervasive probing systems, such as *iPlane* [20] and DIMES [25], exist, but have been designed to predict performance rather than to detect and diagnose faults. Ours is the first study we know of using spoofed packets to determine the direction of path failures, but Govindan and Paxson used them in a similar way to estimate the impact of router processing on measurement tools [10].

Intradomain Troubleshooting: Shaikh and Greenberg [24] proposed to monitor link state announcements within an ISP to identify routing problems. Kompella et al. also developed techniques to localize faults with ISP-level monitoring [15] and used active probing within a tier-1 ISP to detect black holes [14]. Wu et al. [34] used novel data mining techniques to correlate performance problems within an ISP to routing updates. Huang et al. [13] correlated BGP data from an AS with known disruptions; many were detectable only by examining multiple BGP streams.

Our work focuses on a previously unexplored but important design point in the measurement infrastructure space: fine-grained and continuous monitoring of the entire Internet using active probes. It enables fine-grained fault localization, modeling evolution of faults at the level of routers, and comparative evaluation of various resiliency enhancing solutions [1, 12]. Similar in spirit is Teixeira and Rexford's proposal [27], where they argue for each AS to host servers, for distributed monitoring and querying of current forwarding path state. Our work provides less complete information, due to lack of network support, but is easier to deploy. Most similar to us is PlanetSeer, which passively monitors clients of the CoDeeN CDN and launches active probes when it observes anomalies [35]. The focus of their analysis is different, providing complementary results. However, by only monitoring clients, the system covers only 43% of edge ASes and misses entirely any event that prevents a client from connecting to CoDeeN. Furthermore, this represented their aggregate coverage over 3 months, and monitoring stopped if a client had not contacted CoDeeN in 15 minutes, so some ASes may only have been monitored for brief periods. *Hubble*, on the other hand, probes prefixes in 92% of edge ASes every 2 minutes.

7 Conclusion

In this paper, we presented *Hubble*, a system that performs continuous and fine-grained probing of the Internet in order to identify and classify reachability problems in real-time on a global scale. We found that monitoring

of popular BGP feeds alone does not suffice to discover most problems. At the core of our approach is a hybrid monitoring scheme, combining passive BGP monitoring with active probing of the Internet's edge prefix space. We estimate that this approach allows us to discover and monitor 85% of reachability problems, while issuing only 5.5% of the measurement traffic required by a pervasive approach with the same 15-minute granularity. In a three week study conducted with *Hubble*, we identified persistent reachability problems affecting more than 10,000 distinct prefixes, with one in five of the events lasting over 10 hours. Furthermore, two-thirds were cases of partial reachability in which a working physical path demonstrably exists.

Besides identifying problems in real-time across the Internet, we provided important early steps towards classifying problems to aid operators taking corrective action. We identified several hundred prefixes that seem not to be getting the protection that multi-homing is meant to provide; they experienced partial connectivity events where routes terminated in black holes at one provider, but were successful through another. We evaluated a prototype system that uses spoofed probes to solve the difficult problem of differentiating between forward and reverse path failures. In cases to which it fully applied, it worked five times more often than previous techniques. Applying this technique to the multi-homing cases, we isolated the direction of failure for four-fifths of problems and found all to be failures on the forward path to the prefix in question. We believe that in the future we can build on this work to deliver to operators the information they need to dramatically improve global reachability, as well as apply our system to identifying and diagnosing more general performance problems.

Acknowledgments

We gratefully acknowledge Paul Barham, our shepherd, and the anonymous NSDI reviewers for their valuable feedback on earlier versions of this paper. This research was partially supported by the National Science Foundation under grants CNS-0435065, CNS-0519696, and MRI-0619836.

References

- [1] D. G. Anderson, H. Balakrishnan, M. F. Kaawhoek, and R. Morris. Resilient Overlay Networks. In *SOSP*, 2001.
- [2] R. Bush, J. Hiebert, O. Maennel, M. Roughan, and S. Uhlig. Testing the reachability of (new) address space. In *INM*, 2007.
- [3] M. Caesar, L. Subramanian, and R. H. Katz. Root cause analysis of Internet routing dynamics. Technical report, Univ. of California, Berkeley, 2003.
- [4] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *SIGCOMM*, 1988.
- [5] A. Dhamdhare, R. Teixeira, C. Dovrolis, and C. Diot. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT*, 2007.

- [6] N. Feamster, D. G. Andersen, H. Balakrishnan, and M. F. Kaashoek. Measuring the effects of Internet path faults on reactive routing. In *SIGMETRICS*, 2003.
- [7] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.
- [8] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *SIGCOMM*, 2004.
- [9] F. Georgatos, F. Gruber, D. Karrenberg, M. Santcroos, A. Susanj, H. Uijterwaal, and R. Wilhem. Providing active measurements as a regular service for ISPs. In *PAM*, 2001.
- [10] R. Govindan and V. Paxson. Estimating router ICMP generation delays. In *PAM*, 2002.
- [11] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *INFOCOM*, 2000.
- [12] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *OSDI*, 2004.
- [13] Y. Huang, N. Feamster, A. Lakhina, and J. J. Xu. Diagnosing network disruptions with network-wide analysis. In *SIGMETRICS*, 2007.
- [14] R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. Detection and localization of network black holes. In *INFOCOM*, 2007.
- [15] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *NSDI*, 2005.
- [16] N. Kushman, S. Kandula, and D. Katabi. Can you hear me now?! It must be BGP. *CCR*, 37(2):75–84, 2007.
- [17] C. Labovitz, A. Ahuja, and M. Bailey. Shining Light on Dark Address Space. http://www.arbornetworks.com/dmdocuments/dark_address_space.pdf.
- [18] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet routing convergence. In *SIGCOMM*, 2000.
- [19] A. Lambert, M. Meulle, and J.-L. Lutton. Revisiting interdomain root cause analysis from multiple vantage points. In *NANOG*, number 40, June 2007.
- [20] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*, 2006.
- [21] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.
- [22] Z. M. Mao, R. Bush, T. G. Griffin, and M. Roughan. BGP beacons. In *IMC*, 2003.
- [23] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 1997.
- [24] A. Shaikh and A. Greenberg. OSPF monitoring: Architecture, design and deployment experience. In *NSDI*, 2004.
- [25] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. *CCR*, 35(5):31–41, 2005.
- [26] N. Spring, M. Dontcheva, M. Rodrig, and D. Wetherall. How to resolve IP aliases.. Technical report, Univ. of Washington, 2004.
- [27] R. Teixeira and J. Rexford. A measurement framework for pinpointing routing changes. In *ACM SIGCOMM workshop on Network Troubleshooting*, 2004.
- [28] <http://isotf.org/mailman/listinfo/outages>. Outages mailing list.
- [29] <http://www.merit.edu/mail.archives/nanog/>. North American Network Operators Group mailing list.
- [30] <http://www.ripe.net/ris/>. RIPE Routing Information Service.
- [31] <http://www.routeviews.org/>. RouteViews.
- [32] F. Wang, N. Feamster, and L. Gao. Quantifying the effects of routing dynamics on end-to-end Internet path failures. Technical report, Univ. of Massachusetts, Amherst, 2005.
- [33] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush. A measurement study on the impact of routing events on end-to-end Internet path performance. *CCR*, 36(4):375–386, 2006.
- [34] J. Wu, Z. M. Mao, J. Rexford, and J. Wang. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *NSDI*, 2005.
- [35] M. Zhang, C. Zhang, V. S. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, 2004.
- [36] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. An analysis of BGP multiple origin AS (MOAS) conflicts, 2001.

Maelstrom: Transparent Error Correction for Lambda Networks

Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, Einar Vollset
{mahesh, tudorm, ken, hweather, einar}@cs.cornell.edu
Cornell University, Ithaca, NY-14853

Abstract

The global network of datacenters is emerging as an important distributed systems paradigm — commodity clusters running high-performance applications, connected by high-speed ‘lambda’ networks across hundreds of milliseconds of network latency. Packet loss on long-haul networks can cripple application performance — a loss rate of 0.1% is sufficient to reduce TCP/IP throughput by an order of magnitude on a 1 Gbps link with 50ms latency. Maelstrom is an edge appliance that masks packet loss transparently and quickly from inter-cluster protocols, aggregating traffic for high-speed encoding and using a new Forward Error Correction scheme to handle bursty loss.

1 Introduction

The emergence of commodity clusters and datacenters has enabled a new class of globally distributed high-performance applications that coordinate over vast geographical distances. For example, a financial firm’s New York City datacenter may receive real-time updates from a stock exchange in Switzerland, conduct financial transactions with banks in Asia, cache data in London for locality and mirror it to Kansas for disaster-tolerance.

To interconnect these bandwidth-hungry datacenters across the globe, organizations are increasingly deploying private ‘lambda’ networks [35, 39]. Raw bandwidth is ubiquitous and cheaply available in the form of existing ‘dark fiber’; however, running and maintaining high-quality *loss-free* networks over this fiber is difficult and expensive. Though high-capacity optical links are almost never congested, they drop packets for numerous reasons — dirty/degraded fiber [14], misconfigured/malfunctioning hardware [20, 21] and switching contention [27], for example — and in different patterns, ranging from singleton drops to extended bursts [16, 26].

Non-congestion loss has been observed on long-haul networks as well-maintained as Abilene/Internet2 and National LambdaRail [15, 16, 20, 21] — as has its crippling effect on commodity protocols, motivating research into loss-resistant data transfer protocols [13, 17, 25, 38, 43].

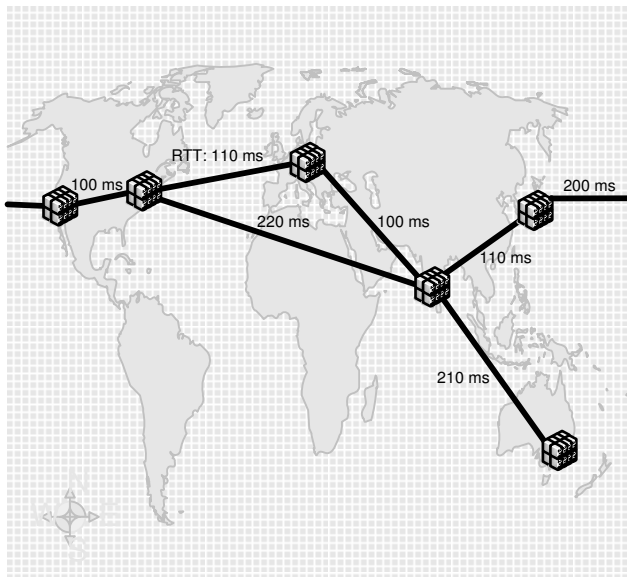


Figure 1: Example Lambda Network

Conservative flow control mechanisms designed to deal with the systematic congestion of the commodity Internet react too sharply to ephemeral loss on over-provisioned links — a single packet loss in ten thousand is enough to reduce TCP/IP throughput to a third over a 50 ms gigabit link, and one in a thousand drops it by an order of magnitude. Real-time applications are impacted by the reliance of reliability mechanisms on acknowledgments and retransmissions, limiting the latency of packet recovery to at least the Round Trip Time (RTT) of the link; if delivery is sequenced, each lost packet acts as a virtual ‘road-block’ in the FIFO channel until it is recovered.

Deploying new loss-resistant protocols is not an alternative in corporate datacenters, where standardization is the key to low and predictable maintenance costs; neither is eliminating loss events on a network that could span thousands of miles. Accordingly, there is a need to *mask* loss on the link, *rapidly* and *transparently*. Rapidly, because recovery delays for lost packets translate into dramatic reductions in application-level throughput; and

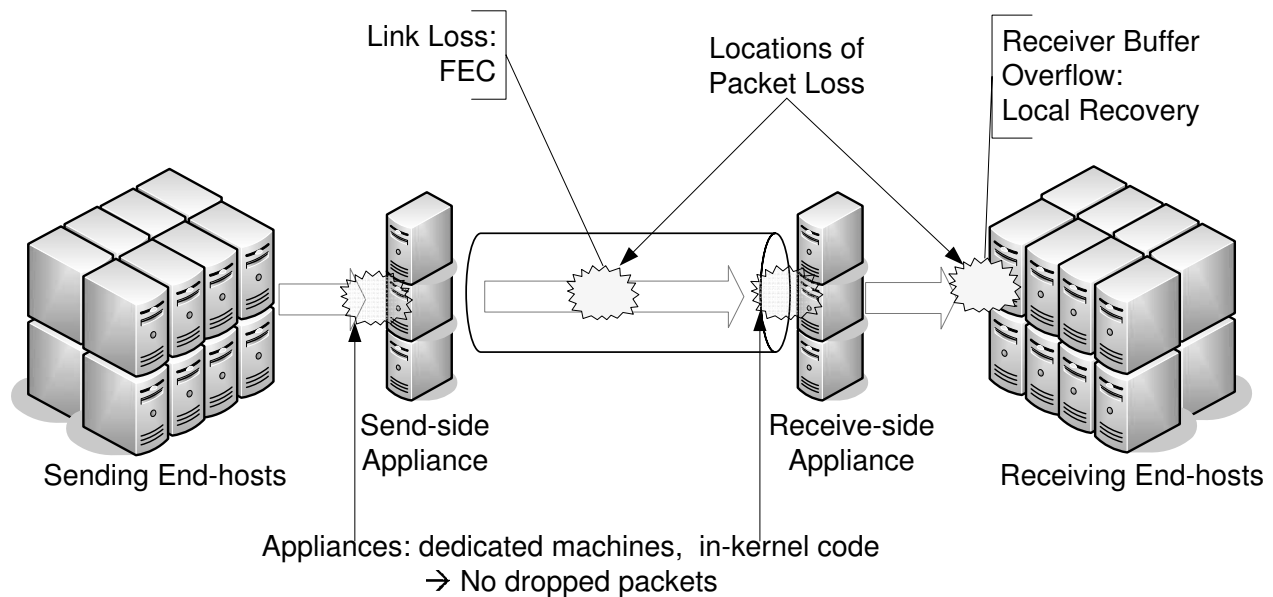


Figure 2: Maelstrom Communication Path

transparently, because applications and OS networking stacks in commodity datacenters cannot be rewritten from scratch.

Forward Error Correction (FEC) is a promising solution for reliability over long-haul links [36] — packet recovery latency is independent of the RTT of the link. While FEC codes have been used for decades within link-level hardware solutions, faster commodity processors have enabled packet-level FEC at end-hosts [18, 37]. End-to-end FEC is very attractive for inter-datacenter communication: it’s inexpensive, easy to deploy and customize, and does not require specialized equipment in the network linking the datacenters. However, end-host FEC has two major issues — First, it’s not transparent, requiring modification of the end-host application/OS. Second, it’s not necessarily rapid; FEC works best over high, stable traffic rates and performs poorly if the data rate in the channel is low and sporadic [6], as in a single end-to-end channel.

In this paper, we present the Maelstrom Error Correction appliance — a rack of proxies residing between a datacenter and its WAN link (see Figure 2). Maelstrom encodes FEC packets over traffic flowing through it and routes them to a corresponding appliance at the destination datacenter, which decodes them and recovers lost data. Maelstrom is completely transparent — it does not require modification of end-host software and is agnostic to the network connecting the datacenter. Also, it eliminates the dependence of FEC recovery latency on the data rate in any single node-to-node channel by encoding over the *aggregated* traffic leaving the datacenter. Finally, Maelstrom uses a new encoding scheme called *layered in-*

terleaving, designed especially for time-sensitive packet recovery in the presence of bursty loss.

The contributions of this paper are as follows:

- We explore end-to-end FEC for long-distance communication between datacenters, and argue that the rate sensitivity of FEC codes and the opacity of their implementations present major obstacles to their usage.
- We propose Maelstrom, a gateway appliance that transparently aggregates traffic and encodes over the resulting high-rate stream.
- We describe *layered interleaving*, a new FEC scheme used by Maelstrom where for constant encoding overhead the latency of packet recovery degrades gracefully as losses get burstier.
- We discuss implementation considerations. We built two versions of Maelstrom; one runs in user mode, while the other runs within the Linux kernel.
- We evaluate Maelstrom on Emulab [45] and show that it provides near lossless TCP/IP throughput and latency over lossy links, and recovers packets with latency independent of the RTT of the link and the rate in any single channel.

2 Model

Our focus is on pairs of geographically distant datacenters that coordinate with each other in real-time. This has long

been a critical distributed computing paradigm in application domains such as finance and aerospace; however, similar requirements are arising across the board as globalized enterprises rely on networks for high-speed communication and collaboration.

Traffic Model: The most general case of inter-cluster communication is one where any node in one cluster can communicate with any node in the other cluster. We make no assumptions about the type of traffic flowing through the link; mission-critical applications could send dynamically generated real-time data such as stock quotes, financial transactions and battleground location updates, while enterprise applications could send VoIP streams, ssh sessions and synchronous file updates between offices.

Loss Model: Packet loss typically occurs at two points in an end-to-end communication path between two datacenters, as shown in Figure 2 — in the wide-area network connecting them and at the receiving end-hosts. Loss in the lambda link can occur for many reasons, as stated previously: transient congestion, dirty or degraded fiber, malfunctioning or misconfigured equipment, low receiver power and burst switching contention are some reasons [14, 20, 21, 23, 27]. Loss can also occur at receiving end-hosts within the destination datacenter; these are usually cheap commodity machines prone to temporary overloads that cause packets to be dropped by the kernel in bursts [6] — this loss mode occurs with UDP-based traffic but not with TCP/IP, which advertises receiver windows to prevent end-host buffer overflows.

What are typical loss rates on long-distance optical networks? One source of information is TeraGrid [5], an optical network interconnecting major supercomputing sites in the US. TeraGrid has a monitoring framework within which ten sites periodically send each other 1 Gbps streams of UDP packets and measure the resulting loss rate [3]. Each site measures the loss rate to every other site once an hour, resulting in a total of 90 loss rate measurements collected across the network every hour. Between Nov 1, 2007 and Jan 25, 2007, 24% of all such measurements were over 0.01% and a surprising 14% of them were over 0.1%. After eliminating a single site (Indiana University) that dropped incoming packets steadily at a rate of 0.44%, 14% of the remainder were over 0.01% and 3% were over 0.1%.

These numbers reflect the loss rate experienced for UDP traffic on an end-to-end path and may not generalize to TCP packets. Also, we do not know if packets were dropped within the optical network or at intermediate devices within either datacenter, though it's unlikely that they were dropped at the end-hosts; many of the measurements lost just one or two packets whereas kernel/NIC losses are known to be bursty [6]. Further, loss occurred on paths where levels of optical link utilization (determined by 20-second moving averages) were consistently

lower than 20%, ruling out congestion as a possible cause, a conclusion supported by dialogue with the network administrators [44].

Other data-points are provided by the back-bone networks of Tier-1 ISPs. Global Crossing reports average loss rates between 0.01% and 0.03% on four of its six inter-regional long-haul links for the month of December 2007 [1]. Qwest reports loss rates of 0.01% and 0.02% in either direction on its trans-pacific link for the same month [2]. We expect privately managed lambdas to exhibit higher loss rates due to the inherent trade-off between fiber/equipment quality and cost [10], as well as the difficulty of performing routine maintenance on long-distance links. Consequently, we model end-to-end paths as dropping packets at rates of 0.01% to 1%, to capture a wide range of deployed networks.

3 Existing Reliability Options

TCP/IP is the default reliable communication option for contemporary networked applications, with deep, exclusive embeddings in commodity operating systems and networking APIs. Consequently, most applications requiring reliable communication over any form of network use TCP/IP.

3.1 The problem with commodity TCP/IP

ACK/Retransmit + Sequencing: Conventional TCP/IP uses positive acknowledgments and retransmissions to ensure reliability — the sender buffers packets until their receipt is acknowledged by the receiver, and resends if an acknowledgment is not received within some time period. Hence, a lost packet is received in the form of a retransmission that arrives no earlier than 1.5 RTTs after the original send event. The sender has to buffer each packet until it's acknowledged, which takes 1 RTT in lossless operation, and it has to perform additional work to retransmit the packet if it does not receive the acknowledgment. Also, any packets that arrive with higher sequence numbers than that of a lost packet must be queued while the receiver waits for the lost packet to arrive.

Consider a high-throughput financial banking application running in a datacenter in New York City, sending updates to a sister site in Switzerland. The RTT value between these two centers is typically 100 milliseconds; i.e., in the case of a lost packet, all packets received within the 150 milliseconds between the original packet send and the receipt of its retransmission have to be buffered at the receiver.

Notice that for this commonplace scenario, the loss of a single packet stops all traffic in the channel to the application for a seventh of a second; a sequence of such

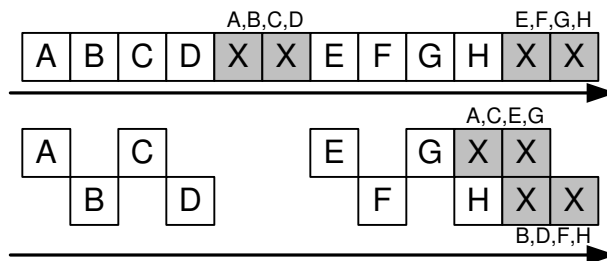


Figure 3: Interleaving with index 2: separate encoding for odd and even packets

blocks can have devastating effect on a high-throughput system where every spare cycle counts. Further, in applications with many fine-grained components, a lost packet can potentially trigger a butterfly effect of missed deadlines along a distributed workflow. During high-activity periods — market crashes at stock exchanges, Christmas sales at online stores, winter storms at air-traffic control centers — overloaded networks and end-hosts can exhibit continuous packet loss, with each lost packet driving the system further and further out of sync with respect to its real-world deadlines.

Sensitive Flow Control: TCP/IP is unable to distinguish between ephemeral loss modes — due to transient congestion, switching errors, or dirty fiber — and persistent congestion. The loss of one packet out of ten thousand is sufficient to reduce TCP/IP throughput to a third of its lossless maximum; if one packet is lost out of a thousand, throughput collapses to a thirtieth of the maximum.

3.2 The Case For (and Against) FEC

FEC encoders are typically parameterized with an (r, c) tuple — for each outgoing sequence of r data packets, a total of $r + c$ data and error correction packets are sent over the channel¹. Significantly, redundancy information cannot be generated and sent until all r data packets are available for sending. Consequently, the latency of packet recovery is determined by the rate at which the sender transmits data. Generating error correction packets from less than r data packets at the sender is not a viable option — even though the data rate in this channel is low, the receiver and/or network could be operating at near full capacity with data from other senders.

FEC is also very susceptible to bursty losses [34]. *Interleaving* [32] is a standard encoding technique used to combat bursty loss, where error correction packets are generated from alternate disjoint sub-streams of data rather than from consecutive packets. For example, with an interleave index of 3, the encoder would create correction packets separately from three disjoint sub-streams: the first containing data packets numbered

$(0, 3, 6 \dots (r - 1) * 3)$, the second with data packets numbered $(1, 4, 7 \dots (r - 1) * 3 + 1)$, and the third with data packets numbered $(2, 5, 8, \dots (r - 1) * 3 + 2)$. Interleaving adds burst tolerance to FEC but exacerbates its sensitivity to sending rate — with an interleave index of i and an encoding rate of (r, c) , the sender would have to wait for $i * (r - 1) + 1$ packets before sending any redundancy information.

These two obstacles to using FEC in time-sensitive settings — rate sensitivity and burst susceptibility — are interlinked through the tuning knobs: an interleave of i and a rate of (r, c) provides tolerance to a burst of up to $c * i$ consecutive packets. Consequently, the burst tolerance of an FEC code can be changed by modulating either the c or the i parameters. Increasing c enhances burst tolerance at the cost of network and encoding overhead, potentially worsening the packet loss experienced and reducing throughput. In contrast, increasing i trades off recovery latency for better burst tolerance without adding overhead — as mentioned, for higher values of i , the encoder has to wait for more data packets to be transmitted before it can send error correction packets.

Importantly, once the FEC encoding is parameterized with a rate and an interleave to tolerate a certain burst length B (for example, $r = 5$, $c = 2$ and $i = 10$ to tolerate a burst of length $2 * 10 = 20$), all losses occurring in bursts of size less than or equal to B are recovered with the same latency — and this latency depends on the i parameter. Ideally, we'd like to parameterize the encoding to tolerate a maximum burst length and then have recovery latency depend on the actual burstiness of the loss. At the same time, we would like the encoding to have a constant rate for network provisioning and stability. Accordingly, an FEC scheme is required where latency of recovery degrades gracefully as losses get burstier, even as the encoding overhead stays constant.

4 Maelstrom Design and Implementation

We describe the Maelstrom appliance as a single machine — later, we will show how more machines can be added to the appliance to balance encoding load and scale to multiple gigabits per second of traffic.

4.1 Basic Mechanism

The basic operation of Maelstrom is shown in Figure 4 — at the send-side datacenter, it intercepts outgoing data packets and routes them to the destination datacenter, generating and injecting FEC repair packets into the stream in their wake. A repair packet consists of a 'recipe' list of data packet identifiers and FEC information generated

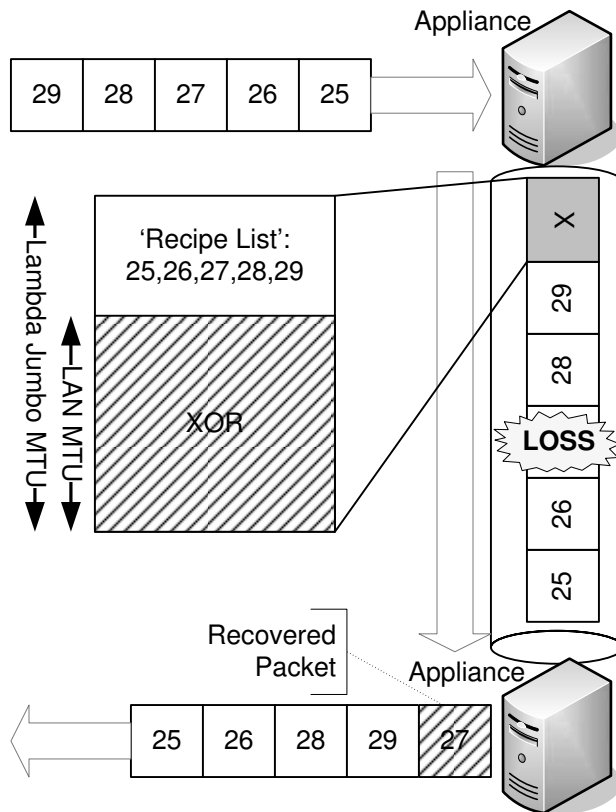


Figure 4: Basic Maelstrom mechanism: repair packets are injected into stream transparently

from these packets; in the example in Figure 4, this information is a simple XOR. The size of the XOR is equal to the MTU of the datacenter network, and to avoid fragmentation of repair packets we require that the MTU of the long-haul network be set to a slightly larger value. This requirement is usually satisfied in practical deployments, since gigabit links very often use ‘Jumbo’ frames of up to 9000 bytes [19] while LAN networks have standard MTUs of 1500 bytes.

At the receiving datacenter, the appliance examines incoming repair packets and uses them to recover missing data packets. On recovery, the data packet is injected transparently into the stream to the receiving end-host. Recovered data packets will typically arrive out-of-order, but this behavior is expected by communication stacks designed for the commodity Internet.

4.2 Flow Control

While relaying TCP/IP data, Maelstrom has two flow control modes: *end-to-end* and *split*. With end-to-end flow control, the appliance routes packets through without modification, allowing flow-control between the end-hosts. In *split* mode, the appliance acts as a TCP/IP

endpoint, terminating connections and sending back ACKs immediately before relaying data on appliance-to-appliance flows; this is particularly useful for applications with short-lived flows that need to ramp up throughput quickly and avoid the slow-start effects of TCP/IP on a long link. The performance advantages of splitting long-distance connections into multiple hops are well known [7] and orthogonal to this work; we are primarily interested in isolating the impact of rapid and transparent recovery of lost packets by Maelstrom on TCP/IP, rather than the buffering and slow-start avoidance benefits of generic performance-enhancing proxies. In the remainder of the paper, we describe Maelstrom with end-to-end flow control.

Is Maelstrom TCP-Friendly? While Maelstrom respects end-to-end flow control connections (or splits them and implements its own proxy-to-proxy flow control as described above), it is not designed for routinely congested networks; the addition of FEC under TCP/IP flow control allows it to steal bandwidth from other competing flows running without FEC in the link, though maintaining fairness versus similarly FEC-enhanced flows [30]. However, friendliness with conventional TCP/IP flows is not a primary protocol design goal on over-provisioned multi-gigabit links, which are often dedicated to specific high-value applications. We see evidence for this assertion in the routine use of parallel flows [38] and UDP ‘blast’ protocols [17, 43] both in commercial deployments and by researchers seeking to transfer large amounts of data over high-capacity academic networks.

4.3 Layered Interleaving

In layered interleaving, an FEC protocol with rate (r, c) is produced by running c multiple instances of an $(r, 1)$ FEC protocol simultaneously with increasing interleave indices $I = (i_0, i_1, i_2 \dots i_{c-1})$. For example, if $r = 8$, $c = 3$ and $I = (i_0 = 1, i_1 = 10, i_2 = 100)$, three instances of an $(8, 1)$ protocol are executed: the first instance with interleave $i_0 = 1$, the second with interleave $i_1 = 10$ and the third with interleave $i_2 = 100$. An $(r, 1)$ FEC encoding is simply an XOR of the r data packets — hence, in layered interleaving each data packet is included in c XORs, each of which is generated at different interleaves from the original data stream. Choosing interleaves appropriately (as we shall describe shortly) ensures that the c XORs containing a data packet do not have any other data packet in common. The resulting protocol effectively has a rate of (r, c) , with each XOR generated from r data packets and each data packet included in c XORs. Figure 5 illustrates layered interleaving for a $(r = 3, c = 3)$ encoding with $I = (1, 10, 100)$.

As mentioned previously, standard FEC schemes can be made resistant to a certain loss burst length at the cost

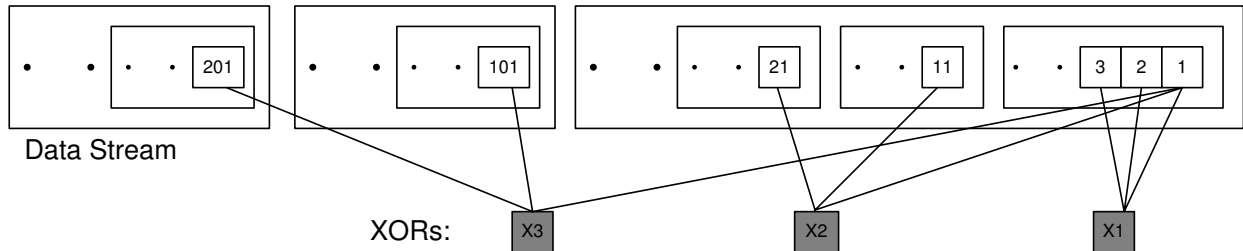


Figure 5: Layered Interleaving: $(r = 3, c = 3), I = (1, 10, 100)$

of increased recovery latency for all lost packets, including smaller bursts and singleton drops. In contrast, layered interleaving provides graceful degradation in the face of bursty loss for constant encoding overhead — singleton random losses are recovered as quickly as possible, by XORs generated with an interleave of 1, and each successive layer of XORs generated at a higher interleave catches larger bursts missed by the previous layer.

The implementation of this algorithm is simple and shown in Figure 6 — at the send-side proxy, a set of repair bins is maintained for each layer, with i bins for a layer with interleave i . A repair bin consists of a partially constructed repair packet: an XOR and the ‘recipe’ list of identifiers of data packets that compose the XOR. Each intercepted data packet is added to each layer — where adding to a layer simply means choosing a repair bin from the layer’s set, incrementally updating the XOR with the new data packet, and adding the data packet’s header to the recipe list. A counter is incremented as each data packet arrives at the appliance, and choosing the repair bin from the layer’s set is done by taking the modulo of the counter with the number of bins in each layer: for a layer with interleave 10, the x th intercepted packet is added to the $(x \bmod 10)$ th bin. When a repair bin fills up — its recipe list contains r data packets — it ‘fires’: a repair packet is generated consisting of the XOR and the recipe list and is scheduled for sending, while the repair bin is re-initialized with an empty recipe list and blank XOR.

At the receive-side proxy, incoming repair packets are processed as follows: if all the data packets contained in the repair’s recipe list have been received successfully, the repair packet is discarded. If the repair’s recipe list contains a single missing data packet, recovery can occur immediately by combining the XOR in the repair with the other successfully received data packets. If the repair contains multiple missing data packets, it cannot be used immediately for recovery — it is instead stored in a table that maps missing data packets to repair packets. Whenever a data packet is subsequently received or recovered, this table is checked to see if any XORs now have singleton losses due to the presence of the new packet and can

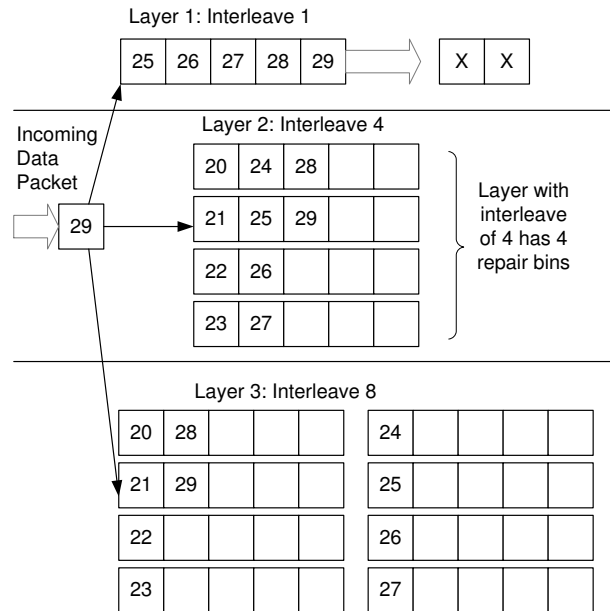


Figure 6: Layered Interleaving Implementation: $(r = 5, c = 3), I = (1, 4, 8)$

be used for recovering other missing packets.

Importantly, XORs received from different layers interact to recover missing data packets, since an XOR received at a higher interleave can recover a packet that makes an earlier XOR at a lower interleave usable — hence, though layered interleaving is equivalent to c different $(r, 1)$ instances in terms of overhead and design, its recovery power is much higher and comes close to standard (r, c) algorithms.

4.3.1 Optimizations

Staggered Start for Rate-Limiting In the naive implementation of the layered interleaving algorithm, repair packets are transmitted as soon as repair bins fill and allow them to be constructed. Also, all the repair bins in a layer fill in quick succession; in Figure 6, the arrival of packets 36, 37, 38 and 39 will successively fill the four repair bins in layer 2. This behavior leads to a large number

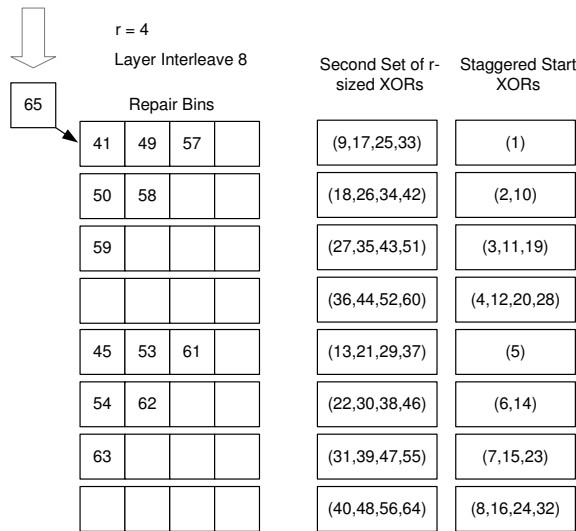


Figure 7: Staggered Start

of repair packets being generated and sent within a short period of time, which results in undesirable overhead and traffic spikes; ideally, we would like to rate-limit transmissions of repair packets to one for every r data packets.

This problem is fixed by ‘staggering’ the starting sizes of the bins, analogous to the starting positions of runners in a sprint; the very first time bin number x in a layer of interleave i fires, it does so at size $x \bmod r$. For example, in Figure 6, the first repair bin in the second layer with interleave 4 would fire at size 1, the second would fire at size 2, and so on. Hence, for the first i data packets added to a layer with interleave i , exactly i/r fire immediately with just one packet in them; for the next i data packets added, exactly i/r fire immediately with two data packets in them, and so on until $r \cdot i$ data packets have been added to the layer and all bins have fired exactly once. Subsequently, all bins fire at size r ; however, now that they have been staggered at the start, only i/r fire for any i data packets. The outlined scheme works when i is greater than or equal to r , as is usually the case. If i is smaller than r , the bin with index x fires at $((x \bmod r) \cdot r/i)$ — hence, for $r = 4$ and $i = 2$, the initial firing sizes would be 2 for the first bin and 4 for the second bin. If r and i are not integral multiples of each other, the rate-limiting still works but is slightly less effective due to rounding errors.

Delaying XORs In the naive implementation, repair packets are transmitted as soon as they are generated. This results in the repair packet leaving immediately after the last data packet that was added to it, which lowers burst tolerance — if the repair packet was generated at interleave i , the resulting protocol can tolerate a burst of i lost data packets excluding the repair, but the burst could swallow both the repair and the last data packet in it as they are not separated by the requisite interleave. The solution to

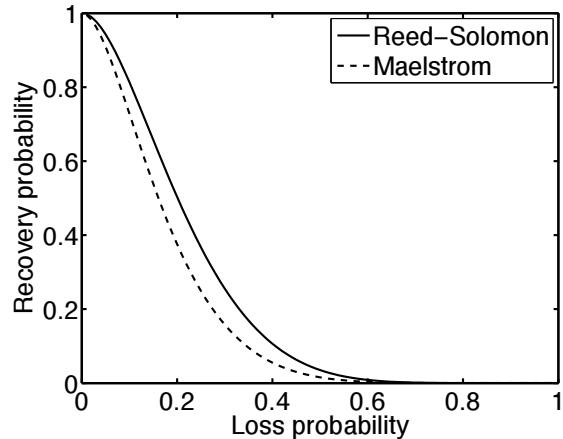


Figure 8: Comparison of Packet Recovery Probability: $r=7$, $c=2$

this is simple — delay sending the repair packet generated by a repair bin until the next time a data packet is added to the now empty bin, which happens i packets later and introduces the required interleave between the repair packet and the last data packet included in it.

Notice that although transmitting the XOR immediately results in faster recovery, doing so also reduces the probability of a lost packet being recovered. This trade-off results in a minor control knob permitting us to balance speed against burst tolerance; our default configuration is to transmit the XOR immediately.

4.4 Back-of-the-Envelope Analysis

To start with, we note that no two repair packets generated at different interleaves i_1 and i_2 (such that $i_1 < i_2$) will have more than one data packet in common as long as the Least Common Multiple (LCM) of the interleaves is greater than $r \cdot i_1$; pairings of repair bins in two different layers with interleaves i_1 and i_2 occur every $LCM(i_1, i_2)$ packets. Thus, a good rule of thumb is to select interleaves that are relatively prime to maximize their LCM, and also ensure that the larger interleave is greater than r .

Let us assume that packets are dropped with uniform, independent probability p . Given a lost data packet, what is the probability that we can recover it? We can recover a data packet if at least one of the c XORs containing it is received correctly and ‘usable’, i.e., all the other data packets in it have also been received correctly, the probability of which is simply $(1-p)^{r-1}$. The probability of a received XOR being unusable is the complement: $(1-(1-p)^{r-1})$.

Consequently, the probability x of a sent XOR being dropped or unusable is the sum of the probability that it was dropped and the probability that it was received and unusable: $x = p + (1-p)(1-(1-p)^{r-1}) = (1-(1-p)^r)$.

Since it is easy to ensure that no two XORs share more than one data packet, the usability probabilities of different XORs are independent. The probability of all the c XORs being dropped or unusable is x^c ; hence, the probability of correctly receiving at least one usable XOR is $1 - x^c$. Consequently, the probability of recovering the lost data packet is $1 - x^c$, which expands to $1 - (1 - (1 - p)^r)^c$.

This closed-form formula only gives us a lower bound on the recovery probability, since the XOR usability formula does not factor in the probability of the other data packets in the XOR being dropped and *recovered*.

Now, we extend the analysis to bursty losses. If the lost data packet was part of a loss burst of size b , repair packets generated at interleaves less than b are dropped or useless with high probability, and we can discount them. The probability of recovering the data packet is then $1 - x^{c'}$, where c' is the number of XORs generated at interleaves greater than b . The formulae derived for XOR usability still hold, since packet losses with more than b intervening packets between them have independent probability; there is only correlation within the bursts, not between bursts.

How does this compare to traditional (r, c) codes such as Reed-Solomon [46]? In Reed-Solomon, c repair packets are generated and sent for every r data packets, and the correct delivery of any r of the $r + c$ packets transmitted is sufficient to reconstruct the original r data packets. Hence, given a lost data packet, we can recover it if at least r packets are received correctly in the encoding set of $r + c$ data and repair packets that the lost packet belongs to. Thus, the probability of recovering a lost packet is equivalent to the probability of losing $c - 1$ or less packets from the total $r + c$ packets. Since the number of other lost packets in the XOR is a random variable Y and has a binomial distribution with parameters $(r + c - 1)$ and p , the probability $P(Y \leq c - 1)$ is the summation $\sum_{z \leq c-1} P(Y = z)$. In Figure 8, we plot the recovery probability curves for Layered Interleaving and Reed-Solomon against uniformly random loss rate, for $(r = 7, c = 2)$ — note that the curves are very close to each other, especially in the loss range of interest between 0% and 10%.

4.5 Local Recovery for Receiver Loss

In the absence of intelligent flow control mechanisms like TCP/IP's receiver-window advertisements, inexpensive datacenter end-hosts can be easily overwhelmed and drop packets during traffic spikes or CPU-intensive maintenance tasks like garbage collection. Reliable application-level protocols layered over UDP — for reliable multicast [6] or high speed data transfer [17], for example — would ordinarily go back to the sender to retrieve the lost packet, even though it was dropped at the receiver after

covering the entire geographical distance.

The Maelstrom proxy acts as a local packet cache, storing incoming packets for a short period of time and providing hooks that allow protocols to first query the cache to locate missing packets before sending retransmission requests back to the sender. Future versions of Maelstrom could potentially use knowledge of protocol internals to transparently intervene; for example, by intercepting and satisfying retransmission requests sent by the receiver in a NAK-based protocol, or by resending packets when acknowledgments are not observed within a certain time period in an ACK-based protocol.

4.6 Implementation Details

We initially implemented and evaluated Maelstrom as a user-space proxy. Performance turned out to be limited by copying and context-switching overheads, and we subsequently reimplemented the system as a module that runs within the Linux 2.6.20 kernel. At an encoding rate of $(8, 3)$, the experimental prototype of the kernel version reaches output speeds close to 1 gigabit per second of combined data and FEC traffic, limited only by the capacity of the outbound network card.

Of course, lambda networks are already reaching speeds of 40 gigabits, and higher speeds are a certainty down the road. To handle multi-gigabit loads, we envision Maelstrom as a small rack-style cluster of blade-servers, each acting as an individual proxy. Traffic would be distributed over such a rack by partitioning the address space of the remote datacenter and routing different segments of the space through distinct Maelstrom appliance pairs. In future work, we plan to experiment with such configurations, which would also permit us to explore fault-tolerance issues (if a Maelstrom blade fails, for example), and to support load-balancing schemes that might vary the IP address space partitioning dynamically to spread the encoding load over multiple machines. For this paper, however, we present the implementation and performance of a single-machine appliance.

The kernel implementation is a module for Linux 2.6.20 with hooks into the kernel packet filter [4]. Maelstrom proxies work in pairs, one on each side of the long haul link. Each proxy acts both as an ingress and egress router at the same time since they handle duplex traffic in the following manner:

- The egress router captures IP packets and creates redundant FEC packets. The original IP packets are routed through unaltered as they would have been originally; the redundant packets are then forwarded to the remote ingress router via a UDP channel.
- The ingress router captures and stores IP packets coming from the direction of the egress router. Upon

receipt of a redundant packet, an IP packet is recovered if there is an opportunity to do so. Redundant packets that can be used at a later time are stored. If the redundant packet is useless it is immediately discarded. Upon recovery the IP packet is sent through a raw socket to its intended destination.

Using FEC requires that each data packet have a unique identifier that the receiver can use to keep track of received data packets and to identify missing data packets in a repair packet. If we had access to end-host stacks, we could have added a header to each packet with a unique sequence number [37]; however, we intercept traffic transparently and need to route it without modification or addition, for performance reasons. Consequently, we identify IP packets by a tuple consisting of the source and destination IP address, IP identification field, size of the IP header plus data, and a checksum over the IP data payload. The checksum over the payload is necessary since the IP identification field is only 16 bits long and a single pair of end-hosts communicating at high speeds will use the same identifier for different data packets within a fairly short interval unless the checksum is added to differentiate between them. Note that non-unique identifiers result in garbled recovery by Maelstrom, an event which will be caught by higher level checksums designed to deal with transmission errors on commodity networks and hence does not have significant consequences unless it occurs frequently.

The kernel version of Maelstrom can generate up to a Gigabit per second of data and FEC traffic, with the input data rate depending on the encoding rate. In our experiments, we were able to saturate the outgoing card at rates as high as (8, 4), with CPU overload occurring at (8, 5) where each incoming data packet had to be XORed 5 times.

4.7 Buffering Requirements

At the receive-side proxy, incoming data packets are buffered so that they can be used in conjunction with XORs to recover missing data packets. Also, any received XOR that is missing more than one data packet is stored temporarily, in case all but one of the missing packets are received later or recovered through other XORs, allowing the recovery of the remaining missing packet from this XOR. In practice we stored data and XOR packets in double buffered red black trees — for 1500 byte packets and 1024 entries this occupies around 3 MB of memory.

At the send-side, the repair bins in the layered interleaving scheme store incrementally computed XORs and lists of data packet headers, without the data packet payloads, resulting in low storage overheads for each layer that rise linearly with the value of the interleave. The

memory footprint for a long-running proxy was around 10 MB in our experiments.

4.8 Other Performance Enhancing Roles

Maelstrom appliances can optionally aggregate small sub-kilobyte packets from different flows into larger ones for better communication efficiency over the long-distance link. Additionally, in split flow control mode they can perform send-side buffering of in-flight data for multi-gigabyte flows that exceed the sending end-host's buffering capacity. Also, Maelstrom appliances can act as multicast forwarding nodes: appliances send multicast packets to each other across the long-distance link, and use IP Multicast [11] to spread them within their datacenters. Lastly, appliances can take on other existing roles in the datacenter, acting as security and VPN gateways and as conventional performance enhancing proxies (PEPs) [7].

5 Evaluation

We evaluated Maelstrom on the Emulab testbed at Utah [45]. For all the experiments, we used a 'dumbbell' topology of two clusters of nodes connected via routing nodes with a high-latency link in between them, designed to emulate the setup in Figure 2, and ran the proxy code on the routers. Figure 10 shows the performance of the kernel version at Gigabit speeds; the remainder of the graphs show the performance of the user-space version at slower speeds. To emulate the MTU difference between the long-haul link and the datacenter network (see Section 4.1) we set an MTU of 1200 bytes on the network connecting the end-hosts to the proxy and an MTU of 1500 bytes on the long-haul link between proxies; the only exception is Figure 10, where we maintained equal MTUs of 1500 bytes on both links.

5.1 Throughput Metrics

Figures 9 and 10 show that commodity TCP/IP throughput collapses in the presence of non-congestion loss, and that Maelstrom successfully masks loss and prevents this collapse from occurring. Figure 9 shows the performance of the user-space version on a 100 Mbps link and Figure 10 shows the kernel version on a 1 Gbps link. The experiment in each case involves running iperf [41] flows from one node to another across the long-distance link with and without intermediary Maelstrom proxies and measuring obtained throughput while varying loss rate (left graph on each figure) and one-way link latency (right graph). The error bars on the graphs to the left are standard errors of the throughput over ten runs; between each run, we flush TCP/IP's cache of tuning parameters to allow for repeatable results. The clients in the experiment are running

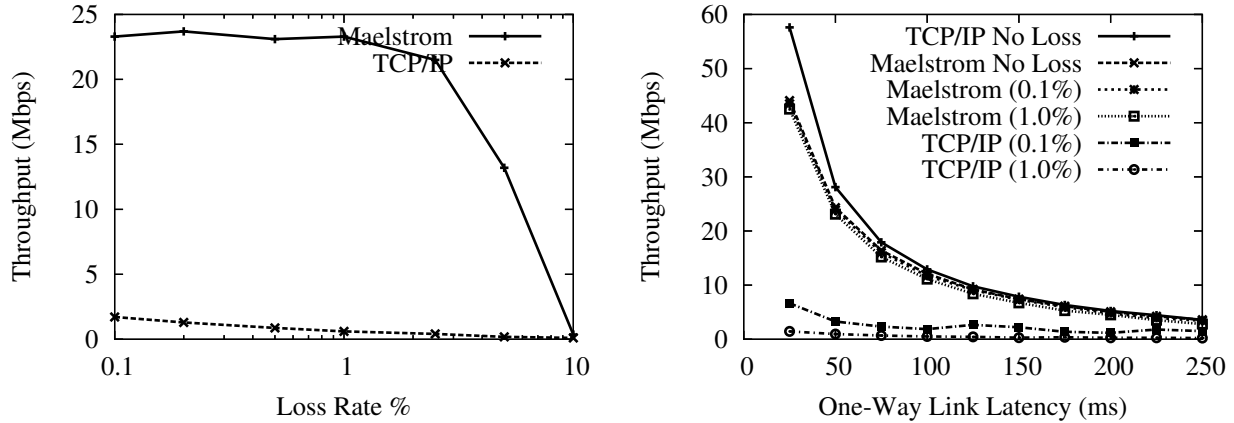


Figure 9: User-Space Throughput against (a) Loss Rate and (b) One-Way Latency

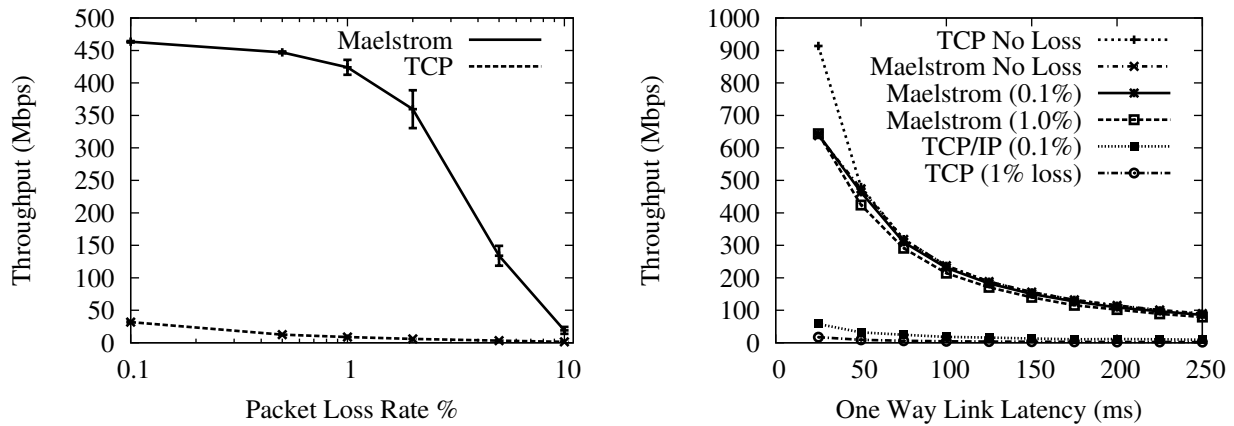


Figure 10: Kernel Throughput against (a) Loss Rate and (b) One-Way Latency

TCP/IP Reno on a Linux 2.6.20 that performs autotuning. The Maelstrom parameters used are $r = 8$, $c = 3$, $I = (1, 20, 40)$.

The user-space version involved running a single 10 second iperf flow from one node to another with and without Maelstrom running on the routers and measuring throughput while varying the random loss rate on the link and the one-way latency. To test the kernel version at gigabit speeds, we ran eight parallel iperf flows from one node to another for 120 seconds. The curves obtained from the two versions are almost identical; we present both to show that the kernel version successfully scales up the performance of the user-space version to hundreds of megabits of traffic per second.

In Figures 9 (Left) and 10 (Left), we show how TCP/IP performance degrades on a 50ms link as the loss rate is increased from 0.01% to 10%. Maelstrom masks loss up to 2% without significant throughput degradation, with the kernel version achieving two orders of magnitude higher throughput than conventional TCP/IP at 1% loss.

The graphs on the right side of Figures 9 and 10

show TCP/IP throughput declining on a link of increasing length when subjected to uniform loss rates of 0.1% and 1%. The top line in the graphs is the performance of TCP/IP without loss and provides an upper bound for performance on the link. In both user-space and kernel versions, Maelstrom masks packet loss and tracks the lossless line closely, lagging only when the link latency is low and TCP/IP's throughput is very high.

To allow TCP/IP to attain very high speeds on the gigabit link, we had to set the MTU of the entire path to be the maximum 1500 bytes, which meant that the long-haul link had the same MTU as the inter-cluster link. This resulted in the fragmentation of repair packets sent over UDP on the long-haul link into two IP packet fragments. Since the loss of a single fragment resulted in the loss of the repair, we observed a higher loss rate for repairs than for data packets. Consequently, we expect performance to be better on a network where the MTU of the long-haul link is truly larger than the MTU within each cluster.

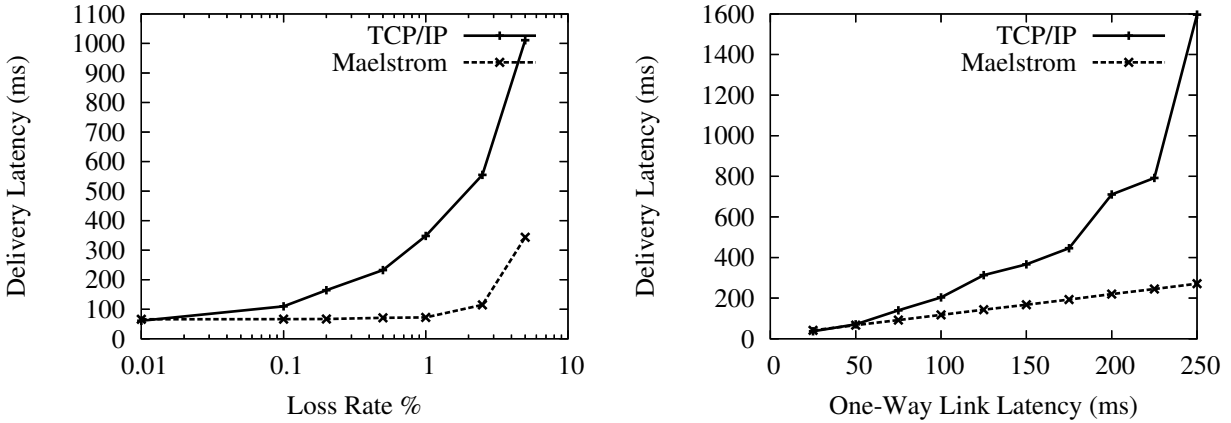


Figure 11: Per-Packet One-Way Delivery Latency against Loss Rate (Left) and Link Latency (Right)

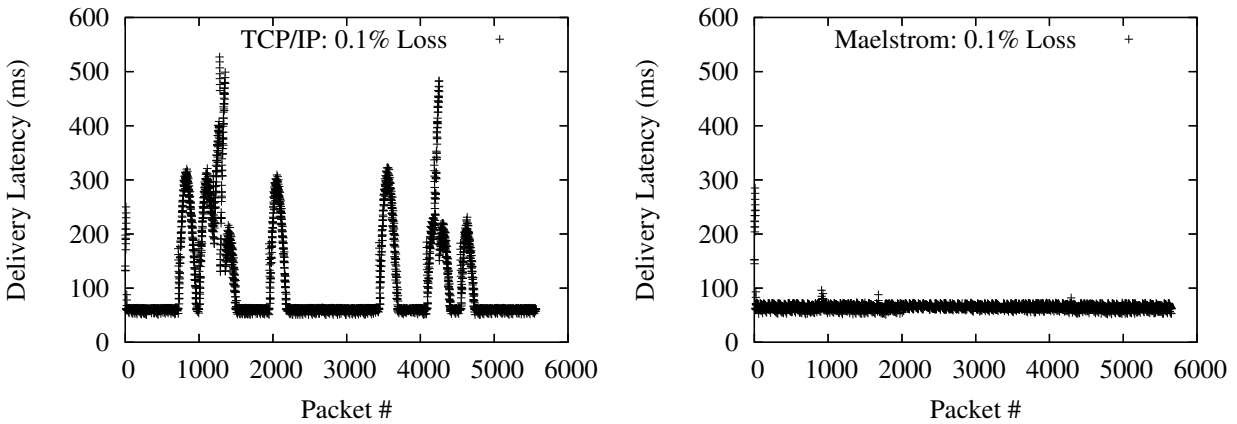


Figure 12: Packet delivery latencies

5.2 Latency Metrics

To measure the latency effects of TCP/IP and Maelstrom, we ran a 0.1 Mbps stream between two nodes over a 100 Mbps link with 50 ms one-way latency, and simultaneously ran a 10 Mbps flow alongside on the same link to simulate a real-time stream combined with other inter-cluster traffic. Figure 11 (Left) shows the average delivery latency of 1KB application-level packets in the 0.1 Mbps stream, as loss rates go up.

Figure 11 (Right) shows the same scenario with a constant uniformly random loss rate of 0.1% and varying one-way latency. Maelstrom’s delivery latency is almost exactly equal to the one-way latency on the link, whereas TCP/IP takes more than twice as long once one-way latencies go past 100 ms. Figure 12 plots delivery latency against message identifier; the spikes in latency are triggered by losses that lead to packets piling up at the receiver.

A key point is that we are plotting the delivery latency of all packets, not just lost ones. TCP/IP delays cor-

rectly received packets while waiting for missing packets sequenced earlier by the sender — the effect of this is shown in Figure 12, where single packet losses cause spikes in delivery latency that last for hundreds of packets. The low data rate in the flow of roughly 10 1KB packets per RTT makes TCP/IP flow control delays at the sender unlikely, given that the congestion control algorithm is Reno, which implements ‘fast recovery’ and halves the congestion window on packet loss rather than resetting it completely [22]. The Maelstrom configuration used is $r = 7, c = 2, I = (1, 10)$.

5.3 Layered Interleaving and Bursty Loss

Thus far we have shown how Maelstrom effectively hides loss from TCP/IP for packets dropped with uniform randomness. Now, we examine the performance of the layered interleaving algorithm, showing how different parameterizations handle bursty loss patterns. We use a loss model where packets are dropped in bursts of fixed length, allowing us to study the impact of burst length on perfor-

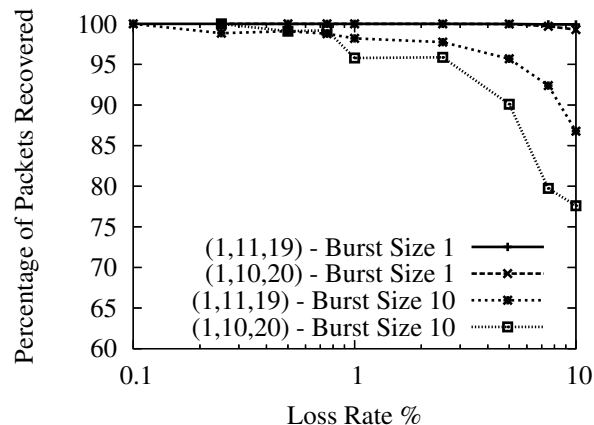


Figure 13: Relatively prime interleaves offer better performance

mance. The link has a one-way latency of 50 ms and a loss rate of 0.1% (except in Figure 13, where it is varied), and a 10 Mbps flow of udp packets is sent over it.

In Figure 13 we show that our observation in Section 4.4 is correct for high loss rates — if the interleaves are relatively prime, performance improves substantially when loss rates are high and losses are bursty. The graph plots the percentage of lost packets successfully recovered on the y-axis against an x-axis of loss rates on a log scale. The Maelstrom configuration used is $r = 8, c = 3$ with interleaves of $(1, 10, 20)$ and $(1, 11, 19)$.

In Figure 14, we show the ability of layered interleaving to provide gracefully degrading performance in the face of bursty loss. On the top, we plot the percentage of lost packets successfully recovered against the length of loss bursts for two different sets of interleaves, and in the bottom graph we plot the average latency at which the packets were recovered. Recovery latency is defined as the difference between the eventual delivery time of the recovered packet and the one-way latency of the link (we confirmed that the Emulab link had almost no jitter on correctly delivered packets, making the one-way latency an accurate estimate of expected lossless delivery time). As expected, increasing the interleaves results in much higher recovery percentages at large burst sizes, but comes at the cost of higher recovery latency. For example, a $(1, 19, 41)$ set of interleaves catches almost all packets in an extended burst of 25 packets at an average latency of around 45 milliseconds, while repairing all random singleton losses within 2-3 milliseconds. The graphs also show recovery latency rising gracefully with the increase in loss burst length: the longer the burst, the longer it takes to recover the lost packets. The Maelstrom configuration used is $r = 8, c = 3$ with interleaves of $(1, 11, 19)$ and $(1, 19, 41)$.

In Figures 16 and 17 we show histograms of recovery

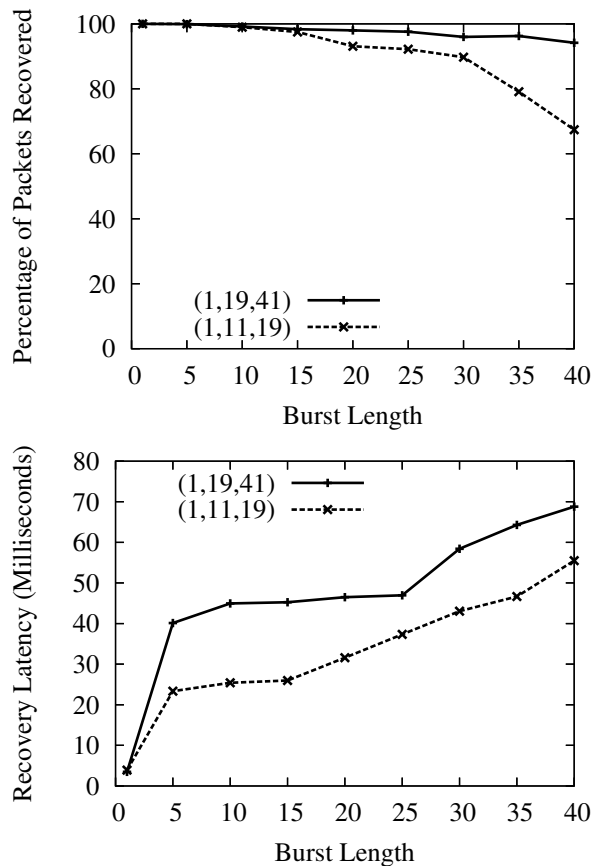


Figure 14: Layered Interleaving Recovery Percentage and Latency

latencies for the two interleave configurations under different burst lengths. The histograms confirm the trends described above: packet recoveries take longer from left to right as we increase loss burst length, and from top to bottom as we increase the interleave values.

Figure 15 illustrates the difference between a traditional FEC code and layered interleaving by plotting a 50-element moving average of recovery latencies for both codes. The channel is configured to lose singleton packets randomly at a loss rate of 0.1% and additionally lose long bursts of 30 packets at occasional intervals. Both codes are configured with $r = 8, c = 2$ and recover all lost packets — Reed-Solomon uses an interleave of 20 and layered interleaving uses interleaves of $(1, 40)$ and consequently both have a maximum tolerable burst length of 40 packets. We use a publicly available implementation of a Reed-Solomon code based on Vandermonde matrices, described in [36]; the code is plugged into Maelstrom instead of layered interleaving, showing that we can use new encodings within the same framework seamlessly. The Reed-Solomon code recovers all lost packets with roughly the same latency whereas layered interleaving re-

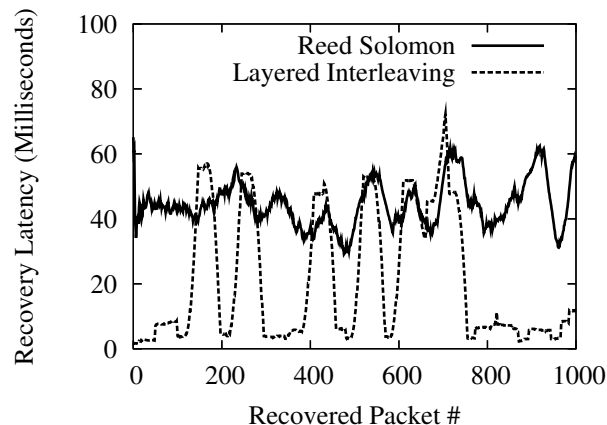


Figure 15: Reed-Solomon versus Layered Interleaving

covers singleton losses almost immediately and exhibits latency spikes whenever the longer loss burst occurs.

6 Related Work

A significant body of work on application and TCP/IP performance over high-speed long-distance networks exists in the context of high-performance computing, grids and e-science. The use of parallel sockets for higher throughput in the face of non-congestion loss was proposed in Pockets [38]. A number of protocols have been suggested as replacements for TCP/IP in such settings — XCP [25], Tsunami [43], SABUL [13] and RBUDP [17] are a few — but all require modifications to end-hosts and/or the intervening network. Some approaches seek to differentiate between congestion and non-congestion losses [8].

Maelstrom is a transparent Performance Enhancing Proxy, as defined in RFC 3135 [7]; numerous implementations of PEPs exist for improving TCP performance on satellite [42] and wireless links [9], but we are not aware of any PEPs that use FEC to mask errors on long-haul optical links.

End-host software-based FEC for reliable communication was first explored by Rizzo [36, 37]. OverQOS [40] suggested the use of FEC for TCP/IP retransmissions over aggregated traffic within an overlay network in the commodity Internet. AFEC [34] uses FEC for real-time communication, modulating the rate of encoding adaptively. The use of end-host FEC under TCP/IP has been explored in [30].

A multitude of different FEC encodings exist in literature; they can broadly be categorized into optimal erasure codes and near-optimal erasure codes. The most well-known optimal code is Reed-Solomon, which we described previously as generating c repair packets from r

source packets; any r of the resulting $r + c$ packets can be used to reconstruct the r source packets. Near-optimal codes such as Tornado and LT [29] trade-off encoding speed for large data sizes against a loss of optimality — the receiver needs to receive slightly more than r source or repair packets to regenerate the original r data packets. Near-optimal codes are extremely fast for encoding over large sets of data but not of significant importance for real-time settings, since optimal codes perform equally well with small data sizes. Of particular relevance are Growth Codes [24], which use multiple encoding rates for different overhead levels; in contrast, layered interleaving uses multiple interleaves for different burst resilience levels without modulating the encoding rate.

The effect of random losses on TCP/IP has been studied in depth by Lakshman [28]. Padhye’s analytical model [33] provides a means to gauge the impact of packet loss on TCP/IP. While most published studies of packet loss are based on the commodity Internet rather than high-speed lambda links, Fraleigh et al. [12] study the Sprint backbone and make two observations that could be explained by non-congestion loss: a) links are rarely loaded at more than 50% of capacity and b) packet reordering events occur for some flows, possibly indicating packet loss followed by retransmissions.

7 Future Work

Scaling Maelstrom to multiple gigabits per second of traffic will require small rack-style clusters of tens of machines to distribute encoding load over; we need to design intelligent load-balancing and fail-over mechanisms for such a scheme. Additionally, we have described layered interleaving with fixed, pre-assigned parameters, and the next step in extending this protocol is to make it adaptive, changing interleaves and rate as loss patterns in the link change.

8 Conclusion

Modern distributed systems are compelled by real-world imperatives to coordinate across datacenters separated by thousands of miles. Packet loss cripples the performance of such systems, and reliability and flow-control protocols designed for LANs and/or the commodity Internet fail to achieve optimal performance on the high-speed long-haul ‘lambda’ networks linking datacenters. Deploying new protocols is not an option for commodity clusters where standardization is critical for cost mitigation. Maelstrom is an edge appliance that uses Forward Error Correction to mask packet loss from end-to-end protocols, improving TCP/IP throughput and latency by orders of magnitude when loss occurs. Maelstrom is easy to install and

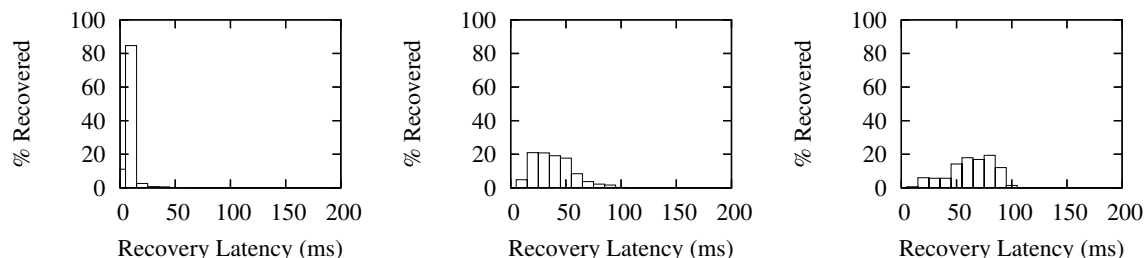


Figure 16: Latency Histograms for $I=(1,11,19)$ — Burst Sizes 1 (Left), 20 (Middle) and 40 (Right)

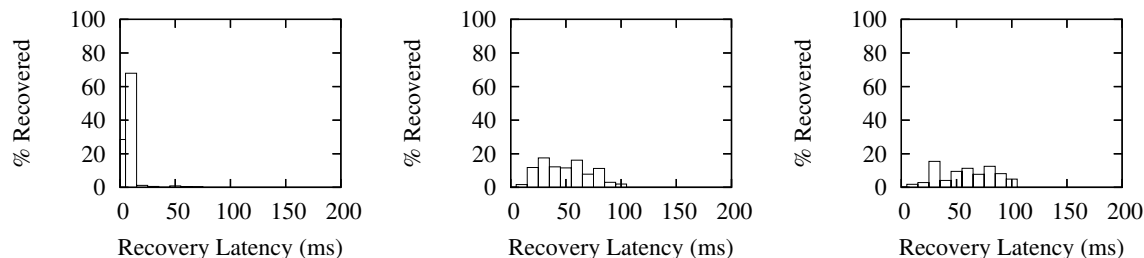


Figure 17: Latency Histograms for $I=(1,19,41)$ — Burst Sizes 1 (Left), 20 (Middle) and 40 (Right)

deploy, and is completely transparent to applications and protocols — literally providing reliability in an inexpensive box.

Acknowledgments

We would like to thank our shepherd Robert Morris and the other reviewers for extensive comments that significantly shaped the final version of the paper. Danny Dolev, Lakshmi Ganesh, T. V. Lakshman, Robbert van Renesse, Yee Jiun Song, Vidhyashankar Venkataraman and Vivek Vishnumurthy provided useful comments. Tom Boures provided valuable insight into the quality of existing fiber links, Stanislav Shalunov provided information on loss rates on Internet2, and Paul Wefel gave us access to TeraGrid loss measurements.

Notes

¹Rateless codes (e.g. LT codes [29]) are increasingly popular and used for applications such as efficiently distributing bulk data [31] — however, it is not obvious that these have utility in real-time communication.

References

- [1] Global crossing current network performance. http://www.globalcrossing.com/network/network_performance_current.aspx. Last Accessed Feb, 2008.
- [2] Qwest ip network statistics. <http://stat.qwest.net/statqwest/statistics.tp.jsp>. Last Accessed Feb, 2008.
- [3] Teragrid udp performance. network.teragrid.org/tgperf/udp/. Last Accessed Feb, 2008.
- [4] Netfilter: firewalling, nat and packet mangling for linux. <http://www.netfilter.org/>, 1999.
- [5] Teragrid. www.teragrid.org, 2008.
- [6] M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Lateral error correction for time-critical multicast. In *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*, 2007.
- [7] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *Internet RFC3135*, June, 2001.
- [8] S. Bregni, D. Caratti, and F. Martignon. Enhanced loss differentiation algorithms for use in TCP sources over heterogeneous wireless networks. In *GLOBECOM 2003: IEEE Global Telecommunications Conference*, 2003.
- [9] R. Chakravorty, S. Katti, I. Pratt, and J. Crowcroft. Flow aggregation for enhanced tcp over wide area wireless. In *INFOCOM*, 2003.
- [10] D. Comer, Vice President of Research and T. Boures, Senior Engineer. Cisco systems, inc. *Private Communication.*, October 2007.
- [11] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.
- [12] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6):6–16, 2003.
- [13] Y. Gu and R. Grossman. SABUL: A Transport Protocol for Grid Computing. *Journal of Grid Computing*, 1(4):377–386, 2003.

- [14] R. Habel, K. Roberts, A. Solheim, and J. Harley. Optical domain performance monitoring. *Optical Fiber Communication Conference*, 2000.
- [15] T. Hacker, B. Athey, and B. Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *IPDPS*, 2002.
- [16] T. J. Hacker, B. D. Noble, and B. D. Athey. The effects of systemic packet loss on aggregate tcp flows. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.
- [17] E. He, J. Leigh, O. Yu, and T. Defanti. Reliable Blast UDP: predictable high performance bulk data transfer. *IEEE International Conference on Cluster Computing*, 2002.
- [18] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.
- [19] J. Hurwitz and W. Feng. End-to-end performance of 10-gigabit Ethernet on commodity systems. *Micro, IEEE*, 24(1):10–22, 2004.
- [20] Internet2. End-to-end performance initiative: Hey! where did my performance go? - rate limiting rears its ugly head. <http://e2epi.internet2.edu/case-studies/UMich/index.html>.
- [21] Internet2. End-to-end performance initiative: When 99% isn't quite enough - educause bad connection. <http://e2epi.internet2.edu/case-studies/EDUCAUSE/index.html>.
- [22] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. *Message to end2end-interest mailing list*, April, 1990.
- [23] L. James, A. Moore, M. Glick, and J. Bulpin. Physical Layer Impact upon Packet Errors. *Passive and Active Measurement Workshop (PAM 2006)*, 2006.
- [24] A. Kamra, J. Feldman, V. Misra, and D. Rubenstein. Growth codes: Maximizing sensor network data persistence. In *Proceedings of ACM Sigcomm*, Pisa, Italy, September 2006.
- [25] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.
- [26] D. Kilper, R. Bach, D. Blumenthal, D. Einstein, T. Landolsi, L. Ostar, M. Preiss, and A. Willner. Optical Performance Monitoring. *Journal of Lightwave Technology*, 22(1):294–304, 2004.
- [27] A. Kimsas, H. Øverby, S. Bjornstad, and V. L. Tuft. A cross layer study of packet loss in all-optical networks. In *AICT/ICIW*, page 65, 2006.
- [28] T. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking (TON)*, 5(3):336–350, 1997.
- [29] M. Luby. LT codes. *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.
- [30] H. Lundqvist and G. Karlsson. TCP with End-to-End Forward Error Correction. *International Zurich Seminar on Communications (IZS 2004)*, 2004.
- [31] P. Maymounkov and D. Mazieres. Rateless codes and big downloads. *IPTPS03*, 2003.
- [32] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.
- [33] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: a simple model and its empirical validation. *SIGCOMM Comput. Commun. Rev.*, 28(4):303–314, 1998.
- [34] K. Park and W. Wang. AFEC: an adaptive forward error correction protocol for end-to-end transport of real-time traffic. *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*, pages 196–205, 1998.
- [35] M. Reardon. Dark fiber: Businesses see the light. http://www.news.com/Dark-fiber-Businesses-see-the-light/2100-1037_3-5557910.html?part=rss&tag=5557910&subj=news.1037.5, 2005. Last Accessed Feb, 2008.
- [36] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, 1997.
- [37] L. Rizzo. On the feasibility of software FEC. *Univ. di Pisa, Italy, January*, 1997.
- [38] H. Sivakumar, S. Bailey, and R. L. Grossman. Psockets: the case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, Washington, DC, USA, 2000. IEEE Computer Society.
- [39] Slashdot.com. Google's secret plans for all that dark fiber. <http://slashdot.org/articles/05/11/20/1514244.shtml>, 2005.
- [40] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz. Overqos: An overlay based architecture for enhancing internet qos. In *NSDI 04: First Usenix Symposium on Networked Systems Design and Implementation*, 2004.
- [41] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf-The TCP/UDP bandwidth measurement tool. <http://dast.nlanr.net/Projects/Iperf>, 2004.
- [42] D. Velenis, D. Kalogeras, and B. Maglaris. SaTPEP: a TCP Performance Enhancing Proxy for Satellite Links. *Proceedings of the Second International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications*, pages 1233–1238, 2002.
- [43] S. Wallace et al. Tsunami File Transfer Protocol. In *PFLD-Net 2003: First Int. Workshop on Protocols for Fast Long-Distance Networks*, 2003.
- [44] P. Wefel, Network Engineer. The University of Illinois' National Center for Supercomputing Applications (NCSA). *Private Communication.*, Feb 2008.
- [45] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.
- [46] S. Wicker and V. Bhargava. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc. New York, NY, USA, 1999.

Swift: A Fast Dynamic Packet Filter

Zhenyu Wu Mengjun Xie Haining Wang
The College of William and Mary
{adamwu, mjxie, hnw}@cs.wm.edu

Abstract

This paper presents *Swift*, a packet filter for high performance packet capture on commercial off-the-shelf hardware. The key features of Swift include (1) extremely low filter update latency for dynamic packet filtering, and (2) Gbps high-speed packet processing. Based on complex instruction set computer (CISC) instruction set architecture (ISA), Swift achieves the former with an instruction set design that avoids the need for compilation and security checking, and the latter by mainly utilizing SIMD (single instruction, multiple data). We implement Swift in the Linux 2.6 kernel for both i386 and x86_64 architectures. The Swift userspace library supports two sets of application programming interfaces (APIs): a BPF-friendly API for backward compatibility and an object oriented API for simplifying filter coding. We extensively evaluate the dynamic and static filtering performance of Swift on multiple machines with different hardware setups. We compare Swift with BPF (the BSD packet filter)—the *de facto* standard for packet filtering in modern operating systems—and hand-coded optimized C filters that are used for demonstrating possible performance gains. For dynamic filtering tasks, Swift is at least three orders of magnitude faster than BPF in terms of filter update latency. For static filtering tasks, Swift outperforms BPF up to three times in terms of packet processing speed, and achieves much closer performance to the optimized C filters.

1 Introduction

A packet filter is an operating system kernel facility that classifies network packets according to criteria given by user applications, and conveys the accepted packets from a network interface directly to the designated application without traversing the network stack. Since the birth of the seminal BSD packet filter (BPF) [14], packet filters have become essential to build fundamental network services ranging from traffic monitoring [12, 21] to network engineering [19] and intrusion detection [20]. In recent years, with dramatically increasing network speed and escalating protocol complexity, packet filters have been facing intensified challenges posed by more dynamic filtering tasks and faster filtering requirements. However, existing packet filter systems have not yet fully addressed these challenges in an efficient and secure manner.

Dynamic filtering tasks refer to on-line packet filtering procedures in which filtering criteria frequently change over time. Typically, when a filtering task cannot fully specify its criteria *a priori* and the unknown part can only be determined at runtime, the filtering criteria have to be updated throughout the filtering process. For example, many network protocols, such as FTP, RTSP (Real Time Streaming Protocol), and SIP (Session Initiation Protocol), establish connections with dynamically-negotiated port numbers. Capturing the network traffic that uses such protocols requires dynamic filtering. Even with pre-determined filter criteria, quite often it is necessary to use dynamic filtering. For instance, a network intrusion detection system (NIDS) needs to perform expensive deep traffic analyses on suspicious network flows. However, applying such costly procedures to every packet in high volume traffic would severely overload the system. Instead, an NIDS could first apply simple filtering criteria, such as monitoring traffic to and from a honeypot or a darknet only. When suspicious activities are detected, the NIDS can then update its filtering criteria and capture the traffic of suspicious hosts for deep inspection.

As the *de facto* packet filter on modern UNIX variants, BPF has shown insufficiency in handling both static and dynamic filtering tasks, particularly the latter [4, 7, 10]. A filter update in BPF must undergo three “pre-processing” phases: compilation, user–kernel copying, and security checking. In the compilation phase, the filtering criteria specified by the human-oriented *pcap* filter language [11] are translated and optimized into the machine-oriented BPF filter program. In the user–kernel copying phase, the compiled filter program is copied into the kernel. Finally, in the security checking phase, the kernel-resident BPF instruction interpreter examines the copied filter program for potentially dangerous operations such as backward branches, ensuring that user-level optimizer errors cannot trigger kernel misbehavior (e.g., infinite loops). Consequently, the whole process of a filter update in BPF induces prolonged latency, which ranges from milliseconds up to seconds depending on criterion complexity and system workload. In high-speed networks, hundreds or even thousands of packets of interest might be missed by BPF during each filter update, effectively leaving a “window of blindness.” Frequent filter updates, often required by a dynamic filtering task,

exacerbate the degree of blindness. A “window of blindness” coinciding with the initialization of a new session can cause serious problems on certain applications such as NIDS, since the beginning of a network connection normally is of particular interest for security analysis [9].

Recent packet filters such as xPF [10] and FFPF [4] move more packet processing capabilities from userspace into the kernel, which reduces context switches and improves overall performance. However, neither filter works well for dynamic filtering tasks. Because xPF uses the BPF-based filtering engine, it offers no improvement on filter update latency. FFPF attempts to solve this problem by using kernel space library functions, called external functions, which are pre-compiled binaries for specific functionalities such as parsing network protocols and updating states. The use of external functions increases filtering speed and eases extensibility, but also increases programming complexity. External functions, unlike safety-checked BPF filters, have access to the kernel’s full privileges. New external functions should thus be carefully examined for potential security bugs, making them a poor fit for frequently-changing dynamic filtering tasks.

In this paper, we propose *Swift*, a packet filter that takes an alternative approach to achieving high performance, especially for dynamic filtering tasks. Like BPF, Swift is based on a fixed set of instructions executed by an in-kernel interpreter. Unlike BPF, Swift is designed to optimize filtering performance with powerful instructions and a simplified computational model. Swift’s instruction set is able to accomplish common filtering tasks with a small number of instructions. These powerful instructions of Swift resemble those in CISC ISAs and support optimizations analogous to SIMD (single instruction, multiple data). Running on the powerful instructions, Swift attains static filtering speedup due mainly to SIMD extension and hierarchical execution optimization, a special runtime optimization technique for avoiding redundant instruction interpretation. More importantly, combining the powerful instructions with the simplified computational model, Swift removes filter compilation and security checking in filter update, and thus significantly improves dynamic filtering performance in terms of filter update latency.

We implement Swift in the Linux 2.6 kernel for both i386 and x86_64 architectures. The kernel implementation of Swift is fully compatible and can coexist with LSF (Linux Socket Filter), “a BPF clone” in Linux. The Swift userspace libraries provide a BPF-friendly application programming interface (API) with textual filter syntax for backward compatibility, and an object-oriented API that simplifies filter coding. To validate the efficacy of Swift, we conduct extensive experiments on multiple machines with different hardware setups and proces-

sor speeds. We compare the performance of Swift with that of LSF and optimized C filters. These C filters are used for demonstrating the possible performance gains obtainable by optimized binary code. For dynamic filtering tasks, Swift achieves at least three orders of magnitude lower filter update latency than LSF, and reduces the number of missing packets per connection by about two orders of magnitude in comparison with LSF. For static filtering tasks with simple filtering criteria, Swift runs as fast as LSF; but with complex filtering criteria, Swift outperforms LSF up to three times in terms of packet processing speed, and performs much closer to the optimized C filters than LSF.

The remainder of this paper is structured as follows. Section 2 surveys related work on packet filters. Section 3 details the design of Swift. Section 4 describes the implementation of Swift. Section 5 evaluates the performance of Swift. Section 6 concludes the paper.

2 Related Work

The CMU/Stanford Packet Filter (CSPF) [16], a kernel-resident network packet demultiplexer, introduces the packet filter concept. CSPF provides a fast path, instead of the normal layered/stacked path, for network packets to reach their destined userspace applications. Thus, the literal meaning of filtering in CSPF is *delayed demultiplexing* [25]. The original motivation behind CSPF is to facilitate the implementation of network protocols such as TCP/IP at userspace. Although the purposes and techniques vary in subsequent packet filters, the CSPF model of kernel-resident and protocol-independent packet filtering is inherited by all its descendants.

BPF [14] aims to support high-speed network monitoring applications such as `tcpdump` [11]. Users inform the in-kernel filtering machine of their interests through a predicate-based filtering language [15], and then receive from BPF the packets that conform to filtering criteria. To achieve high performance, BPF introduces in-place packet filtering to reduce unnecessary cross-domain copies, a register-based filter machine to fix the mismatch between the filter and its underlying architecture, and a control flow graph (CFG) model to avoid redundant computations. BPF+ [3] further enhances the performance of BPF by exploiting global data-flow optimization to eliminate redundant predicates across filter criteria and employing just-in-time compilation to convert a filtering criterion to native machine code. xPF [10] increases the computation power of BPF by using persistent memory and allowing backward jumps.

In response to `tcpdump`’s inefficiency at handling dynamic ports, a special monitoring tool `mmdump` [24] has been developed to capture Internet multimedia traffic, in which dynamic ports are widely used. `mmdump` reduces the cost of compilation by exploiting the uniformity of

its filtering criterion patterns. A customized function in `mmdump` assembles new filtering criteria by using pieces of pre-compiled criterion blocks preserved from the initial filter compilation. Swift's high-level SIMD instructions and hierarchical instruction optimization can be viewed as a generalization of this technique, but Swift's techniques apply to any type of filter and require no special compiler techniques.

MPF (Mach Packet Filter) [26], PathFinder [2], and DPF (Dynamic Packet Filter) [8] are filters designed to demultiplex packets for user-level networking. To efficiently demultiplex packets for multiple user-level applications and to dispatch fragmented packets, MPF extends the instruction set of BPF with an associative match function. With the same goal of achieving high filter scalability as MPF, PathFinder, abstracts the filtering process as a pattern matching process and adopts a special data structure for the abstraction. The abstraction makes PathFinder amenable to implementation in both software and hardware and capable of handling Gbps network traffic. DPF utilizes dynamic code generation technology, instead of a traditional interpreter-based filter engine, to compile packet filtering criteria into native machine code. DPF-like dynamic code generation could improve Swift's performance on static filtering tasks.

The Fairly Fast Packet Filter (FFPF, later renamed as Streamline) [4] is the most recent research on packet filtering. Unlike traditional packet filters such as BPF, FFPF is a framework for network monitoring. Within the framework of FFPF, multiple packet filtering programs can be simultaneously loaded into the kernel. The processing flow among these programs is organized as a directed graph. In comparison to the filtering architecture of BPF, FFPF can significantly reduce the cost of packet copying for multiple concurrent filtering programs by using flow group, shared circular buffers, and even hardware (e.g., Network Processing Unit). Therefore, FFPF achieves far greater scalability than BPF. FFPF expands filter capacity via external functions, which are essentially native code running in kernel space. In addition, FFPF features language neutral design and provides backward compatibility with BPF.

FFPF and Swift are complementary as they target quite different problems. FFPF focuses on the packet filtering framework and its main contribution lies in the improvement of scalability for supporting multiple concurrent monitoring applications, while Swift aims at the packet filtering engine and provides a fast, flexible, and safe filtering mechanism for individual applications. By virtue of the language neutral design of FFPF, Swift can be implemented within the FFPF framework, taking maximal advantage of both designs.

Besides software-based packet capture solutions, multiple hardware-based solutions [5, 18] have been pro-

posed to meet the challenge posed by extremely high speed networks. Specifically, FPGAs and ASICs have been widely used in recent intrusion detection and prevention systems [9, 22]. Moreover, other than the packet-filter-based network monitoring architecture, there exist many specialized-architecture monitoring systems such as OC3MAN [1], Windmill [13], Nprobe [17], and SCAMPI [6]. Even with these hardware or specialized system solutions, packet filtering still plays a major role in network monitoring and measurement due to its simplicity, universal installation, high cost-effectiveness, and rich applications.

3 Design

In this section, we first present the motivation of Swift and its design overview, then we detail the design of Swift, including its unique ISA, and finally we analyze the characteristics of Swift in terms of performance and security.

3.1 Motivation

The inefficiency of BPF observed in our past experiences directly motivated Swift's design. The most significant performance degradation of BPF occurs in dynamic filtering tasks. This degradation is mainly caused by frequent filter updates. As mentioned above, the unduly long filter update latency in BPF is attributed to three filter pre-processing operations: filter re-compilation, user-kernel copying, and security checking. While the latter two play non-negligible roles in the long delay, the majority of the latency is introduced by filter re-compilation [7, 24]. The duration of a filter update in BPF would be significantly shortened if the re-compilation were selectively performed only on the changed primitive, or totally skipped, as `mmdump` accomplishes for selected filtering tasks. However, for general purpose network monitoring tasks, re-compiling the entire BPF filter is inevitable for each update, because its instruction set architecture and filter program organization are unsuitable for fast update.

BPF uses a RISC-like instruction set for a low-level register machine abstraction. Thus, each *pcap* language primitive is translated into an instruction block that comprises a *variable* number of simple instructions. Changing a primitive in a filter often alters the size of the corresponding instruction block. Without re-compiling, we need to modify code-offset-related instructions (e.g., conditional branch) throughout the entire compiled filter to accommodate the change. Control flow optimization, which is indispensable for BPF to speed up filter execution, makes the matter even worse. The BPF control flow optimization merges multiple identical instructions into one. This significantly reduces both filter program size and execution time, but complicates updates to instructions shared by several primitives.

In addition to filter update latency, we also find that the filter execution efficiency of BPF can be improved substantially. The RISC-like ISA in BPF induces high instruction interpretation overhead. Interpretation overhead refers to the operations an interpreter must perform before executing an actual filter instruction, such as program counter maintenance, instruction loading, operation decoding, and so forth. Those operations are unproductive towards evaluating filter criteria, but cannot be omitted. Because each BPF instruction accomplishes merely a very simple task, such as loading, arithmetic, and conditional branching, most of the CPU time in executing a BPF program is spent uselessly as interpretation overhead, and the CPU time spent in evaluating the actual packet filtering criteria only makes up a small fraction of the total. Our measurement results (detailed in Section 5.2) show that the BPF interpretation overhead is about 5.2 nanoseconds on average in a machine with Intel 64-bit Xeon 2.0GHz CPU, and makes up nearly 56% of the average instruction execution time.

BPF’s continuing widespread use can be mainly attributed to (1) the generic pseudo-machine abstraction, which guarantees cross-platform compatibility, and (2) its natural-language-like, primitive based filter language, which ensures ease of use to application developers and network administrators. Therefore, we decide to inherit from BPF the pseudo-machine abstraction and language primitives, while developing our own filtering model to achieve significant performance improvement.

3.2 Design Overview

The primary objective of Swift is to achieve low filter update latency. Our approach to reaching this goal is by reducing filter criterion pre-processing on filter update as much as possible. More specifically, we attempt to avoid filter re-compilation and optimization, allow “in-place” filter updating, and eliminate security checking.

To achieve “compilation free” update, filtering criteria must map *directly* onto interpreter instructions. This makes a high-level, CISC-like instruction set architecture a natural choice for Swift. In addition to saving compilation time, the CISC-like instruction set also opens a door for performance optimization. A complex Swift instruction is able to accomplish the same task as several simple BPF instructions, thereby reducing instruction interpretation overhead.

Two design choices are made to enable in-place filter modification: fixing instruction length and removing filter optimization. By fixing the instruction length, we avoid the need to shift instructions on instruction replacement. By removing filter optimization, not only do we save precious time during a filter update, but also preserve one-to-one mapping between filtering primitives and filter program instructions: no instructions are

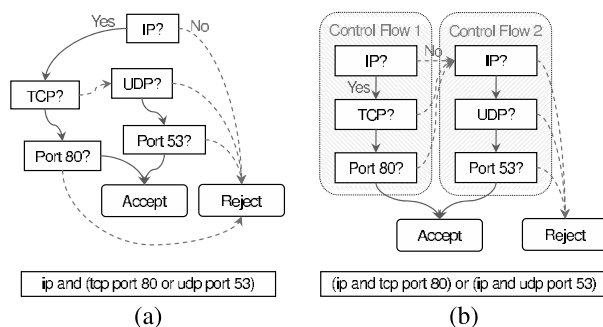


Figure 1: Filter organizations of BPF (a) and Swift (b) for criteria matching HTTP and DNS traffic

shared. As a result, updates to a filter can be directly applied to the affected instructions without altering other instructions or filter program structure. This feature further helps to optimize filter update by reducing unnecessary user–kernel data copying. Only the updated part of a filter criterion is copied from userspace to the kernel.

A simplified computational model ensures filter program safety for Swift without security checking. With the specialized ISA, each Swift instruction is able to perform a set of complex pattern matching operations. The execution path of a filter program is determined by the Boolean evaluation result of each instruction: either continue (“true”) or abort (“false”). Therefore, Swift does not need storage or branch instructions to control the execution path of a filter program. With a fixed set of instructions, acyclic execution path, and zero data storage, any Swift filter program is safe to run in the kernel.

Our secondary objective is to increase filter execution efficiency. We achieve this goal by exploring the following two optimizations: SIMD expansion to the Swift instruction set and hierarchical execution optimization. SIMD allows an interpreter to perform a single instruction interpretation and apply the same operation on many sets of data, thereby significantly reducing the cost of instruction interpretation. SIMD has been widely used in contemporary high performance processors, such as Intel Pentium series and IBM Power series processors. While Swift’s design ensures low filter update latency, it also forfeits the benefit associated with filter program optimization. To offset the possible performance loss, we introduce an alternative optimization method called hierarchical execution optimization. This optimization is based on our observation that during a dynamic filtering process, the newly-added primitives are often related to some existing ones. For example, the new primitives quite often monitor the same host but on different ports or capture the same protocol traffic but for different hosts. Therefore, the existing primitives can be viewed as the “parent” of the new primitives. Swift utilizes this hierar-

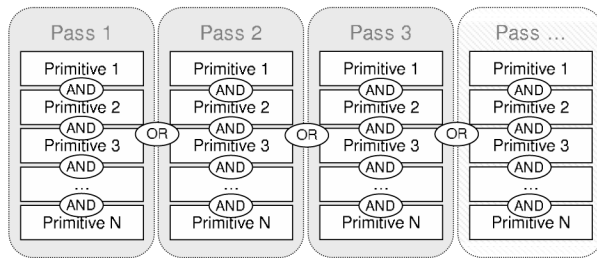


Figure 2: Swift filter structure

chical relationship among primitives to avoid redundant instruction executions. Instead of actively optimizing filter programs, i.e., performing automatic optimization in filter update, Swift makes the primitive hierarchy a hint for the filter execution engine, and leaves applications responsible for constructing the hierarchy.

3.3 Detailed Design

The Boolean logic in a Swift filter is organized in disjunctive normal form. Figure 1 (b) illustrates the control flow organization of Swift, while the control flow graph of BPF, which is semantically equivalent, is shown in Figure 1 (a) for comparison. In the Swift control flow organization, each disjunct cluster—the rounded rectangle with shaded background—specifies a complete set of primitives that a packet must satisfy in order to be accepted by the Swift filter. In Swift, we name such a disjunct cluster a *Pass*, meaning a “passage” of packets.

A pass consists of one or more literals. In a BPF filter, a literal corresponds to a *pcap* language primitive. Swift inherits the primitives from BPF. However, instead of realizing a primitive with multiple simple instructions, Swift maps each type of primitives into a pseudo-machine instruction—the basic building block of a filter program. Figure 2 illustrates the structure of a Swift filter.

3.3.1 Swift Instruction Set

All Swift instructions have the same size, facilitating fast instruction modification on filter update. The Swift instruction layout is shown in Figure 3, where one 32-bit command field is followed by seven 32-bit parameter fields. Such a nicely-aligned 32-byte structure ensures efficient memory accesses.

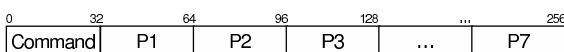


Figure 3: The Swift instruction format

We formulate our instruction set based on BPF primitives. We first classify BPF primitives into two categories according to their addressing modes. “Direct addressing” primitives, such as “ether proto” and “ip src host,” fetch

data from an absolute offset in a packet. “L1 indirect addressing” primitives, such as “tcp dst port,” address data by calculating the variable header length of a protocol layer and adding a relative offset to it. We then generalize the manipulation and comparison operations used in the semantics of BPF primitives. There are three types of basic operations: (1) test if equal, (2) mask and test if equal, and (3) test if in range. Each type also has some variations on operand width (short or long integer). Finally, we design the complex instructions to accomplish the corresponding operations. We come up with 14 different operations that, alone or by combination, are able to perform equivalent operations of any BPF primitives except “expr,” which involves arbitrary arithmetic.

To further exploit the CISC architecture and enhance performance, we introduce a new addressing mode, “L2 indirect addressing,” with four additional instructions. In the new addressing mode, filtering operations address data by first performing “L1 indirect addressing” to retrieve the related information, which is used to calculate the variable header length of a deeper layer, and then adding the relative offset. While BPF does not provide such primitives, there are practical demands such as filtering based on TCP payload. Moreover, we add four more “power instructions” that perform equivalent operations of several frequently-used BPF primitive combinations, such as “ip src and dst net” and “tcp src or dst port.” Therefore, in total Swift has 22 different types of instructions.

Table 1 lists a selection of Swift instructions, which captures the characteristics of Swift’s CISC-like ISA. The four columns from leftmost to rightmost refer to the addressing mode, the instruction type, the instruction functionality, and the equivalent BPF operation(s), respectively. Swift instructions are fairly generic in that given different parameters, one Swift instruction can function as several different *pcap* language primitives. Examples are given in the fourth column. Based on the Swift instruction set, we can derive alternative faster implementations for some BPF primitives. For instance, the “ip and tcp port” primitive in BPF requires three initial steps with six instructions to examine whether a packet is IP, non-fragment, and TCP. In Swift, we can take advantage of the “continuous masked comparison” instruction (D_LEQ_M), to perform the same examination in a single instruction.

We add the SIMD feature into the Swift instruction set by packing additional operands into unused parameter fields. For instance, the “Direct addressing load, test if equal” instruction (D_EQ) uses only one 32-bit operand. In contrast, the SIMD version of this instruction can carry up to six additional operands, and the corresponding operation becomes “Direct addressing load, and test if equal to any of P[1] through P[7].”

Table 1: Sample of Swift instruction set

Addressing mode	Opcode	Semantics	Corresponding BPF operation
Direct Addressing	D_EQ	Compare a 32bit integer at an absolute offset to a supplied integer operand	Compare host IP address
	D_MEQ	Similar to the above, but a bitmask is applied to both comparands before comparison	Compare packet protocol; Compare host IP netmask
	D_LEQ_M	Compare up to three continuous 32bit integers at an absolute offset to the supplied operands (each pair of comparands is associated to a bitmask)	Compare source and destination IP netmask; Tell if a packet is IP, non-fragment, and TCP/UDP
Layer 1 Indirect Addressing	L1_SEQ	Compare a 16bit shortint at an indirect offset to a supplied shortint operand	Compare IP protocol; Compare TCP / UDP port
	L1_SRNG	Test if a 16bit shortint at an indirect offset is within a numerical range specified in the parameter list	Capture TCP / UDP packets on a range of ports
Layer 2 Indirect Addressing	L2_LEQ	Compare up to five continuous 32bit integers at an indirect offset to the supplied operands	TCP payload content matching
	L2_LEQ_M	Compare up to two continuous 32bit integers at an indirect offset to the supplied operands (each pair of comparands is associated to a bitmask)	TCP payload content matching (bit precision)

3.3.2 Swift Pass and Filter Program

A series of instructions connected by logic “AND” form a pass. When a packet arrives, the instructions of a pass are evaluated one by one. If all evaluation results are “true,” the packet is accepted and copied to userspace. Otherwise, if any evaluation result is “false,” the packet fails the current pass, and will be tested by remaining passes or dropped if it fails all passes. Passes are thus effectively independent and are combined by logic “OR.”

We achieve the hierarchical execution optimization feature by establishing hierarchical relationships among passes. When a Swift filter is initially set, the passes it contains are created by the application from scratch. In subsequent changes, if a new pass is related to one of the existing passes, the application is entitled to add the new pass by duplicating an existing pass and modifying the copy, or “child” pass. Performing duplication, instead of creating a pass afresh, has two benefits. First, the application saves time in updating the criterion—only the difference between the old and new control flows needs to be updated. Second, the parent–child relationship is noted by the filtering engine and is used to optimize filter execution.

When a pass is added by duplication, Swift makes a bit-exact copy of the parent pass and then marks all the instructions of the child pass as “copied,” a hint for the filtering engine that the marked instruction is exactly the same as the corresponding one in its parent pass. When an instruction in the child pass is later modified, the associated “copied” mark is removed. To evaluate a Swift filter, the filtering engine traverses the passes according to their hierarchical relationships (if any) in a depth-first manner. A parent pass is evaluated before its children. If the parent pass succeeds, then, as for any pass, the filtering engine halts. Otherwise, Swift records those instructions that succeeded. When evaluating child passes,

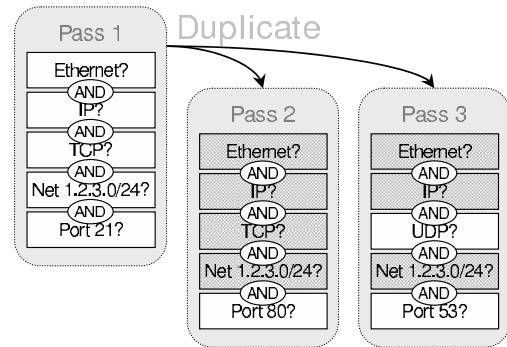


Figure 4: Hierarchical pass relation diagram

Swift need not re-execute any copied instruction that succeeded in the parent.

Figure 4 illustrates an example of the pass relation in a Swift filter. Pass 1 is created from scratch, while the other two passes are added by duplicating pass 1. The instructions bearing the “copied” mark have shaded background, so the filtering engine may skip their evaluations.

3.4 Analysis

Before giving detailed analysis of Swift in terms of performance and security, we first summarize the shared design principles of Swift with other packet filters, especially BPF, as well as its unique design features that distinguish Swift from other packet filters. The shared features are marked with “+,” and the unique ones marked with “◇.”

- + Runs as a kernel module, filtering packets in place.
- + Uses architecture-independent pseudo-machine.
- ◇ Utilizes CISC ISA with SIMD support.
- ◇ Enables compile-free, in-place filter modification.
- ◇ Ensures security with simplified computational model.

3.4.1 Performance

The performance superiority of Swift mainly originates from two aspects: high filter execution efficiency and low filter update latency.

Using a complex instruction set and SIMD benefits static filtering performance. Thanks to the capability of aggregating multiple simple operations into one instruction, a Swift program has much fewer instructions than its BPF counterpart. As a result, even though its per-instruction interpretation overhead is slightly higher than that of BPF, Swift achieves much lower interpretation overhead of an entire filter program. While the filter engine size of Swift (24KB) is much larger than that of BPF (6KB), our experimental results show that the larger code size has insignificant impact on performance. Even running Swift on the CPU with only 12KB L1 cache (“PC1” in our experiment setup), there is still no observable performance degradation indicating cache thrashing.

Swift’s superior dynamic filtering performance is mainly attributed to its very low filter update latency. For a filtering program with N primitives and experiencing M changes per update, the three pre-processing phases in BPF—recompiling the entire filter, copying the whole compiled filter code to the kernel, and security checking—all take $O(N)$ runtime. However, performing the same filter update in Swift involves neither compilation nor security checking. The only required operations, mapping the changed primitives into instruction opcodes and parameters, and copying the modified instructions into kernel, take $O(M)$ runtime. Because the filter update in Swift is only related to the number of changes per update (M), not to the complexity of the existing filter (N), its filter update latency can be substantially lower than that of BPF, especially when N is large (i.e., the filtering criteria are complicated). Furthermore, hierarchical execution optimization can avoid performance degradation caused by redundant filter instructions.

3.4.2 Security

Security, and filter code safety in particular, have always been a concern in packet filter design. Since modern packet filters execute in kernel space, without proper code safety checking, a faulty filter program containing infinite loops, wild jumps, out-of-bound array indexes, etc., could lead to unpredictable results. In addition, a maliciously crafted filter program can bypass any user-level access protection and can seriously undermine system security.

Depending on the design model, different packet filters have different mechanisms to enforce the security of the filter programs. The FFPF filter languages allow memory allocation, and hence, FFPF has compile-time checks to control resource consumption and run-time checks to detect array boundary violations. In contrast,

BPF only needs to perform a security check in the kernel just before the filter program is attached; any program containing backward or out-of-bound jumps or illegal instructions is rejected. However, Swift enforces security in its design and eliminates the necessity for run-time security checks. Swift trades off some of its computational power, i.e., arbitrary data manipulation, for a simpler computational model. Because any Swift program is an acyclic DFA (deterministic finite-state automaton), the interpreter is always in a pre-determined state, the execution of a finite-size filter program is always bounded, and Swift requires no security check at all.

Two rationales justify this design tradeoff. First, the reduction of computational power is harmless in the context of packet filtering. A packet filter is a very specific system tool with a well defined set of operations. PathFinder [2] shows that all operations in packet filtering can be generalized as pattern matching. The *pcap* filter language uses the special primitive “*expr*” to support arbitrary data manipulation. However, this primitive is rarely used in practice, because its main usage is to specify uncommon filtering criteria that are not covered by regular primitives. Second, BPF’s support for arbitrary data manipulation comes with a high cost of its performance. Instead of following BPF’s strategy, we apply the strategy of “*optimize for the common case and prepare for the worst*” in Swift’s design.

BPF does not differentiate predefined and arbitrary data manipulation operations. Instead, BPF executes any data manipulation by breaking it down to multiple elementary operations. Thus, BPF wastes a significant amount of time in interpretation, and sometimes it takes longer time to interpret an instruction than to execute it. Swift supports well defined and commonly used data manipulations by incorporating each variant in a single instruction, and integrating their implementations into the filtering engine. Since those operations are carried out by native binary code, Swift achieves very high execution efficiency. Swift cannot perform data manipulations that are not defined in its instruction set. Instead, the user applications need to carry out the custom data manipulations by themselves. However, in case an unsupported data manipulation is desperately needed, for example, when a new protocol requires a different data manipulation, we can always add new instructions to Swift.

4 Implementation

We have implemented the Swift kernel engine and userspace libraries in Linux 2.6. Currently we provide implementations for both i386 and x86_64 architectures, and we plan to port Swift to other open-source UNIX variants such as FreeBSD in the future.

Table 2: Selection of *libswift* APIs

Routine	Functionality
SPF_Open	Create and attach an empty Swift filter to a socket
SPF_NewPass	Create a new pass in the Swift filter
SPF_DelPass	Remove a pass from the Swift filter
SPF_DupPass	Create a copy of a given pass
SPF_SelectOp	Assign a predefined operation (equivalent to a <i>pcap</i> language primitive) to an <i>instruction</i> of a given pass
SPF_AddParam	Add an additional (SIMD) parameter to an <i>instruction</i> in the given pass
SPF_DelParam	Remove a given parameter from an <i>instruction</i> in the given pass
SPF_PokeInst	Change an arbitrary part of an <i>instruction</i> in the given pass

4.1 Kernel Implementation

Swift coexists with the Linux kernel’s LSF (Linux Socket Filter). LSF is the module equivalent to BPF in BSD UNIX and the default packet filtering module for the widely used *libpcap* library. Our implementation requires little modification to the existing kernel code, and is compatible with the existing packet filtering framework. Swift’s user–kernel communication mechanism uses the `setsockopt()` system call. Swift filter programs are attached to the same kernel data structure `sk_filter` as LSF filter programs, with a flag set to tell two kinds of programs apart. Packets captured by Swift and LSF share the same delivery path no matter which packet filter is being used.

4.2 Userland Libraries

The *libpcap* library provides a set of well-designed routines for setting filter programs and processing packets, as well as utility functions for handling devices and dumping captured packets. Instead of hacking *libpcap* to incorporate Swift, we developed a set of complementary libraries. Applications based on Swift can seamlessly use those *libpcap* functions that are unrelated to filter setting, but must invoke a separate mechanism to communicate with the Swift filter engine for filter program installation and update.

As shown in Table 2, the C library *libswift* provides a set of function APIs for the convenient manipulation of Swift filter programs. We also implement a C++ library *ooswift*, providing object-oriented filter program control and manipulation and improved debugging support. Table 3 shows a common filtering criterion expressed in *pcap* primitives, in Swift using *ooswift* and in compiled LSF code. The table illustrates the clear logical connection and easy transformation between the Swift filter program and the *pcap* language primitives.

Table 3: A filter expressed by *pcap*, Swift, and LSF

Filtering criterion by <i>pcap</i>				
ip src net 192.168.254.0/24 and tcp dst port 23				
Swift filter program				
Pass.Inst(0)→EtherIP_TCP_NonFrag() Pass.Inst(1)→Ether_IPSrc(0xFFFFFFFF00, 0xC0A8FE00) Pass.Inst(2)→EtherIP_TCUDPDst(23)				
LSF filter program				
(00)	ldh	[12]		
(01)	jeq	#0x800	jt 2	jf 13
(02)	ld	[26]		
(03)	and	#0xffffffff00		
(04)	jeq	#0xc0a8fe00	jt 5	jf 13
(05)	ldb	[23]		
(06)	jeq	#0x6	jt 7	jf 13
(07)	ldh	[20]		
(08)	jseq	#0x1fff	jt 13	jf 9
(09)	ldxb	4*([14]&0xf)		
(10)	ldh	[x + 16]		
(11)	jeq	#0x17	jt 12	jf 13
(12)	ret	#96		
(13)	ret	#0		

5 Evaluation

In this section, we evaluate the performance of Swift on both dynamic and static filtering tasks and compare it with that of LSF and C kernel filters. The C kernel filters (“Opt-C” for short in the following) are hand-coded, compiled (using `gcc -O2` option) C programs that provide some indication of possible performance gains obtainable by non-interpreted binary code. Each C kernel filter is coded for a specific filtering task. We use the performance of Opt-C filters as an approximation to the performance of FFPF filters. An FFPF filter written in FPL-3, which is FFPF’s native language, is first transformed into a C program and compiled into native code by `gcc`, and then loaded as a kernel module for use. Since both FPL-3 filters and our Opt-C filters run inside the kernel natively and only a single filter program runs in each experiment, the performance difference between FFPF filters and Opt-C filters should be minimal.

Swift, LSF, and Opt-C filters share the same filtering framework and only differ in filtering engine. Therefore,

Table 4: Testbed machine configurations

	CPU	L2	FSB
PC1	1 × Intel Pentium 4 2.8GHz (32-bit)	1MB	533MHz
PC2	2 × Intel Xeon 2.8GHz (32-bit w/ HyperThreading)	512KB	800MHz
PC3	2 × Intel Xeon 2.0GHz (EM64T DualCore)	4MB	1333MHz
PCS	1 × Intel Pentium D 2.8GHz (EM64T DualCore)	2MB	800MHz

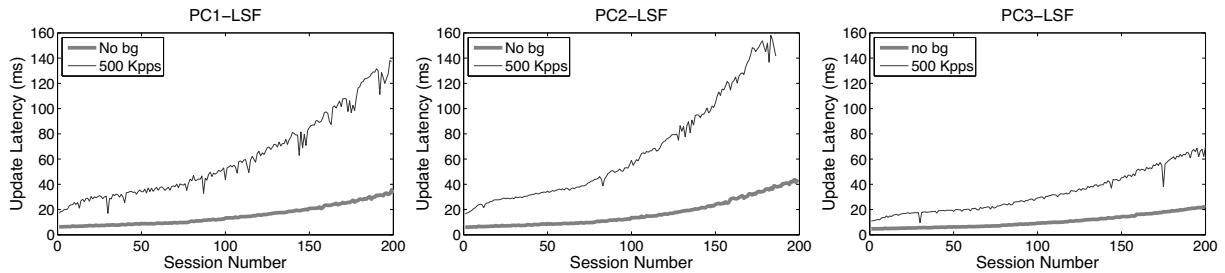


Figure 5: LSF filter update latency

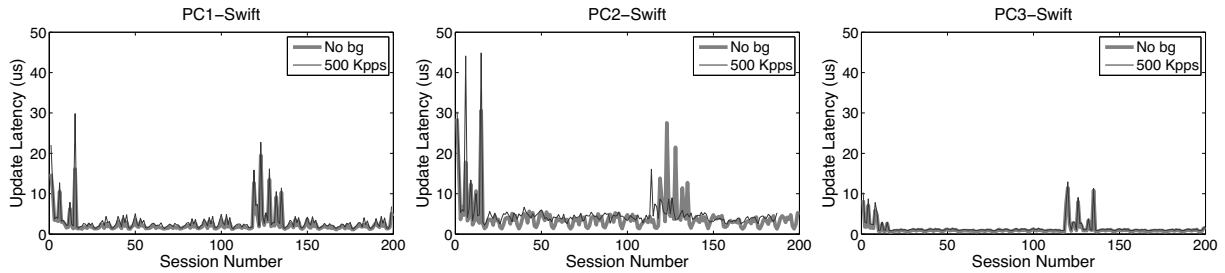


Figure 6: Swift filter update latency

we use the number of CPU cycles spent by the filtering engine as the micro-benchmark metric. This metric is computed by taking the difference of the x86 Time-Stamp Counter (TSC) just before and right after a specific filter operation. For dynamic filtering tasks, the operation is filter update, while for static filtering tasks, the operation is filter evaluation. In order to compare filter performance across different platforms, we further convert CPU cycles into clock time based on the corresponding machine’s processor frequency.

To evaluate the filters in a realistic but controllable environment, we set up a testbed using a Gbps SMC managed switch to connect four different machines. We use the mirror function of the switch to redirect the traffic on the specified source port to the mirror port. The packet generator machine (PCS) connected to the source port replays traces, and one of the other three machines (PC1–3) connected to the mirror port captures the replayed traffic as a monitoring device. The four machines (PCS and PC1–3) have different generations of processors ranging from Pentium 4 32-bit to the latest Xeon dual core 64-bit. The configurations of these machines are listed in Table 4.

5.1 Dynamic Filtering Performance

We use the task of capturing FTP passive mode traffic, a typical dynamic filtering task, to measure the performances among Swift, LSF, and Opt-C filters. We developed an application called *FTPCap* to monitor FTP traffic and collect performance statistics. Three variants that use Swift, LSF, or Opt-C are called *FTPCap-Swift*, *FTPCap-LSF*, and *FTPCap-Opt-C*, respectively.

5.1.1 Experimental Setup

In passive mode FTP, the server port of a control connection is fixed (usually 21), but the server ports of data connections are dynamically assigned. *FTPCap-LSF* initially employs “(ip and tcp port ftp)” to capture FTP control packets. When a control packet containing the port number for a new data connection is captured, the server IP address and port number for the new connection will be recorded, and *FTPCap-LSF* will generate a new criterion similar to “(ip and tcp port ftp) or (ip x1 and (tcp port y1 or tcp port y2)) or (ip x2 and (tcp port y3 or tcp port y4))”, in which “x1” and “x2” refer to the server IP addresses and “y1 . . . y4” refer to the port numbers. The LSF optimizer performs better when the port numbers of the same server are grouped together. Correspondingly, *FTPCap-Swift* initializes the first pass with the criterion “(ip and tcp port ftp)” to capture FTP control packets. When a data connection setup event is detected, *FTPCap-Swift* either simply includes the new port number in the corresponding pass if the server is already observed, or adds a pass using hierarchical execution optimization otherwise. *FTPCap-Opt-C*, unlike the previous two, has no code for filter setting and updating because the work is already taken by the *Opt-C* filter. It simply turns on/off the *Opt-C* filter.

The FTP traffic trace is obtained in a LAN environment. We set up 10 FTP servers with different IP addresses. For each server we make 20 concurrent passive-mode file transfer connections, which are initiated one by one. In other words, at maximum there are 200 concurrent passive FTP data connections. This trace lasts

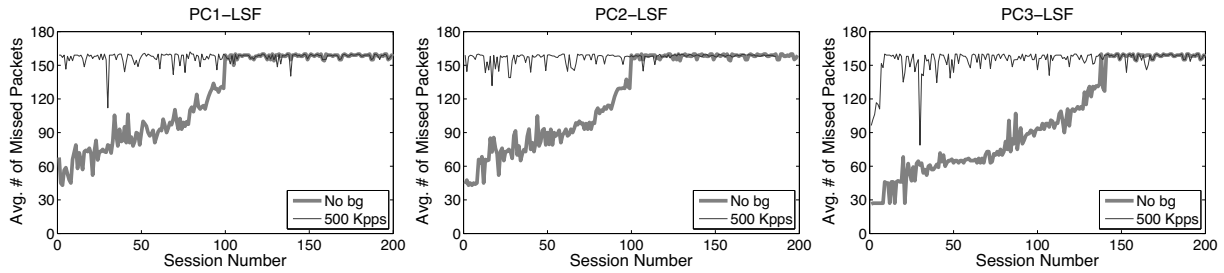


Figure 7: Missing packets per data connection by LSF

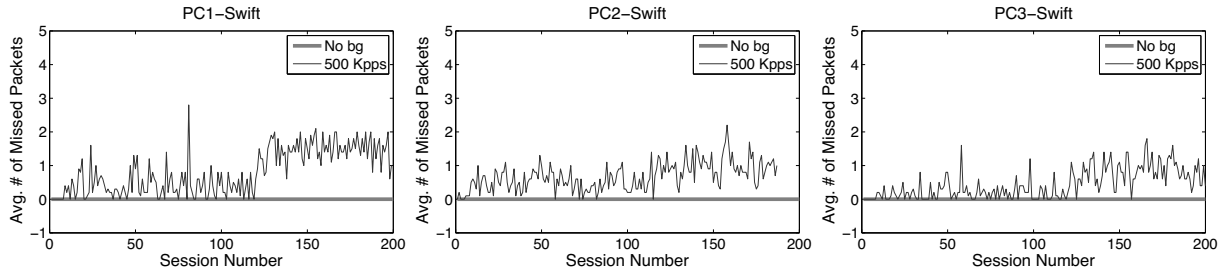


Figure 8: Missing packets per data connection by Swift

45 seconds with 3,948 packets per second (pps) on average. In addition, we emulate the scenario of monitoring FTP packets under high-rate background traffic by mixing the captured FTP traffic with a constant high-rate (500 Kpps) non-FTP background traffic. The background traffic is generated by using `tcpreplay` [23] to play back a large trace file, which is captured at the edge gateway of our campus network.

Besides using filter update latency as the micro-benchmark performance metric, we also use the number of missing packets per data connection as the macro-benchmark performance metric. The missing packets refer to those packets that are not captured by FTPCap at the beginning of a newly-established data connection. The packet miss is caused by the filter update latency being larger than the FTP client acknowledgment delay—the interval between the time when the client receives the port assignment message and the time when the client starts to communicate with the server on that port. The metric is derived by counting the number of the transmitted packets (including TCP control packets) prior to the first packet captured by FTPCap, based on the offline analysis of the replayed trace.

5.1.2 Experimental Results

We run FTPCap-LSF and FTPCap-Swift 20 times each on PC1–3, 10 times with the FTP traffic trace replayed and the other 10 times with the mixed traffic trace replayed. We take the median of 10 experimental results as the final result. FTPCap-Opt-C is also tested. Because there is no filter update at userspace, its filter update latency is zero and no packet is missed by FTPCap-Opt-

C for either FTP traffic or mixed traffic. Therefore, we focus on the performance comparison between LSF and Swift.

Figures 5 and 6 show how filter update latency changes with an increase in concurrent data connections for LSF and Swift, respectively. The thick and thin curves show the filter update latencies for the traces with no background traffic and with 500 Kpps background traffic, respectively. The most significant difference between Figures 5 and 6 lies in the scale of the y-axis. While the filter update latency for LSF is on the order of milliseconds (ms), the filter update latency for Swift is only on the order of microsecond (μ s). By eliminating filter compilation and security checking, Swift gains at least three orders of magnitude speedup against LSF in filter update. Over 99% of LSF’s latency is caused by user-level filter recompilation, but the remaining user-kernel copy and security check latency is still much larger than Swift’s entire update latency. For example, the user-kernel copy and security check latency on PC3 grows from 10μ s to 20μ s in the experiment. FTPCap running on PC2 does not capture all control packets that carry dynamic port information under mixed traffic, which results in incomplete thin curves in “PC2-LSF” and “PC2-Swift.” The missing critical control packets are mainly due to PC2’s insufficient processing capacity.

As shown in Figure 5, both concurrent connections and background traffic affect the filter update latency of LSF. When the number of concurrent connections increases, the filtering criterion expressed in *pcap* language becomes longer. And the compilation procedure and security checking consume more CPU cycles. In contrast,

Table 5: Static filters in *pcap* language and their instruction counts in LSF and Swift

Filter	Description	LSF Inst.#	Swift Inst.#
1	"" (Accept all packets)	1	0
2	"ip"	3	1
3	"ip src net 192.168.2.0/24 and dst net 10.0.0.0/8"	10	2
4	"ip src or dst net 192.168.2.0/24"	10	2
5	"ip and tcp port (ssh or http or imap or smtp or pop3 or ftp)"	23	2
6	"ip and (not tcp port (80 or 25 or 143) and not ip host (...))" (The ellipsis mark stands for 38 IP addresses ORed together.)	95	10

the filter update latency of Swift is basically insusceptible to changes in concurrent connections and background traffic: although all Swift curves in Figure 6 fluctuate slightly, the thin curves overlap with the thick curves to a great extent. This is because Swift filter updates are incremental and adding filter instructions for a new connection takes almost constant time. The large spikes of Swift curves, which occur at the beginning and around the addition of the 120th connection, are attributed to the relatively large overheads caused by pass duplication.

Figures 7 and 8 illustrate the average number of missing packets per data connection by LSF and Swift, respectively. The y-axis scales are again significantly different. The average numbers of missing packets per connection for LSF range from 30 to 160, while those for Swift are only one or two at maximum. Without background traffic, Swift does not miss any packet no matter how many concurrent connections exist. With background traffic, the average levels of the "500 Kpps" curves slightly lift after around 120 concurrent connections, which coincides with the occurrence of the second group of large spikes in Figure 6. The lift of fluctuation level may be attributed to the added passes and related pass duplication. The addition of more passes extends the filtering path for non-FTP packets and results in more CPU time spent on non-FTP traffic filtering. Even so, Swift only misses one or two packets per connection.

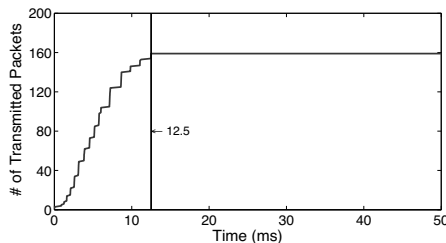


Figure 9: Initial transmission of a data connection

There are two additional issues associated with the LSF curves in Figure 7. First, the ceiling phenomena—both “No bg” and “500 Kpps” curves bounded by 160—are caused by the rate-limiting of the FTP servers. As illustrated by Figure 9, in the initial period of a data con-

nection, the servers first transmit about 160 packets in tens of milliseconds and then stay idle for the next several hundred milliseconds (not all shown) to limit downloading rate. Such bounding behavior occurs for a wide range of rate-limit settings (e.g., from 100 KBps to 2 MBps). Since an LSF filter update latency is always shorter than the duration of the idle phase, the number of missing packets in each update is bounded. Second, the round-trip time (RTT) of the FTP trace is small, varying from tens of microseconds to hundreds of microseconds, as the trace is collected in a LAN environment. A larger RTT would cause fewer packets to be transmitted during the time window of a filter update, thus reducing the impact of filter updates on packet missing. Compared to LSF, Swift is almost insensitive to the variation of RTT, and hence can support applications that require high-fidelity data capture in diverse network environments.

5.2 Static Filtering Performance

We use six sets of filters with increasing complexity, as shown in Table 5, for static filtering performance evaluation. The instruction numbers of these filters in LSF and Swift are also listed for comparison. The Opt-C filter programs show performance gains that could potentially be achieved by improving LSF and Swift to native code speeds.

The trace for static filtering is captured at the gateway of our campus network. It contains over 14 million packets (75 bytes snap length) and its size is around 1.1GB. We play back the trace file at 250 Kpps rate using *tcpreplay*. Assuming an average of 500 bytes per packet, the playback rate represents a fully utilized 1Gbps link bandwidth. We record the average time spent in accepting and rejecting packets separately, and select the larger value of the two as the filter performance data. We choose the larger value, instead of the smaller one or the average, because the worse case runtime is much less affected by network traffic conditions, such as traffic speed and composition.

Figure 10 illustrates the per-packet processing time of LSF, Swift, and Opt-C on all machines for each filter. In addition, Table 6 details the breakdown of the per-packet processing time for both LSF and Swift

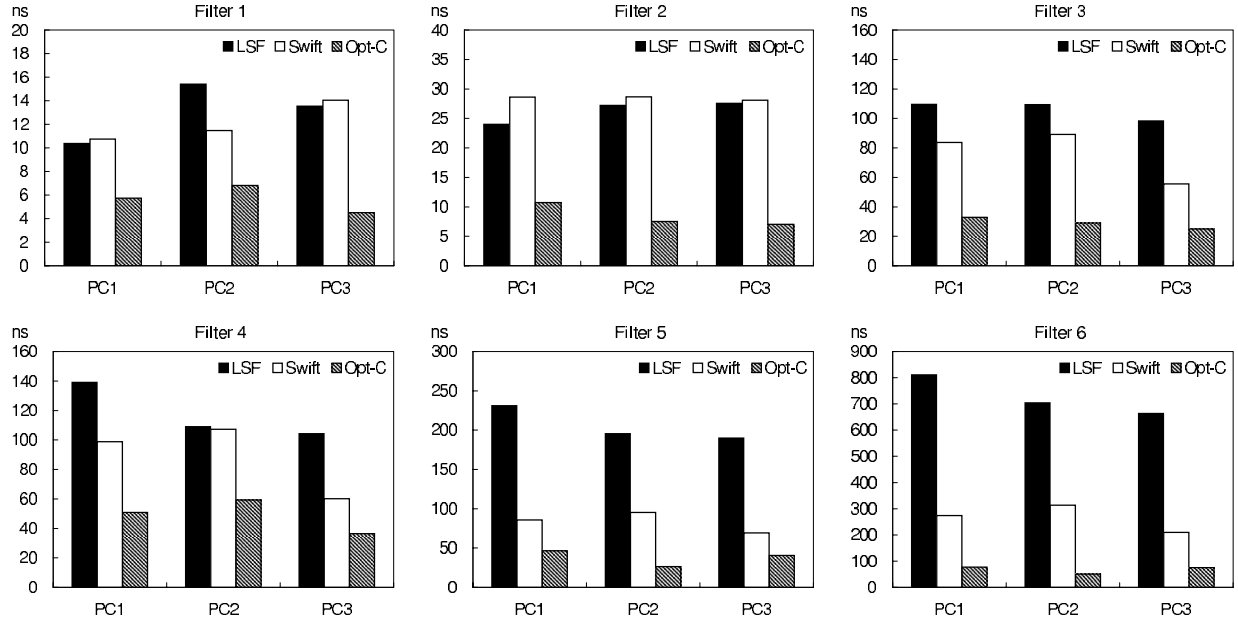


Figure 10: Per-packet processing time of each static filter (nanoseconds)

on PC3. The “Exec.” column shows the average execution time per instruction and the average number of instructions executed per packet, in the format of (time/instruction) \times (instruction count). The “Aux.” column shows the auxiliary processing time spent on filter engine setup and shutdown operations, such as call/ret instructions and local stack maintenance.

Filters 1 and 2 are the simplest criteria designed to show the minimum overhead induced by the filtering engine. The corresponding results in Figure 10 demonstrate that Swift and LSF have approximately the same processing speed with these two simple filters. Both Swift and LSF run slower than Opt-C. Table 6 further sheds some light on the performances of both LSF and Swift filter engines. For filter 1, the LSF filter program only consists of a simple “ret” instruction, and thus the 5.2 nanoseconds per-instruction execution time is mainly determined by LSF’s interpretation overhead. In contrast, the Swift filter engine is designed to accept all packets by default. Therefore, the Swift filter program does not contain any code, and its processing time is spent entirely on the filter engine setup and shutdown. By adding a “nop” instruction for Swift to execute before accepting a packet, we estimate Swift’s interpretation overhead to be about 8.2 nanoseconds. For filter 2, although the per-instruction execution time of LSF is 29% shorter than that of Swift, its overall execution time is longer than that of Swift. This is because the instruction count ratio between LSF and Swift is three to one.

Filters 3 and 4 are light load criteria designed to demonstrate filtering engine performance on basic packet

Table 6: Processing time breakdown for LSF and Swift

Filter	LSF		Swift	
	Exec.	Aux.	Exec.	Aux.
1	5.2ns \times 1.0	13.8ns	(N/A) \times 0	23.1ns
2	10.3ns \times 3.0	14.1ns	14.5ns \times 1.0	22.1ns
3	9.4ns \times 9.0	12.9ns	13.8ns \times 2.0	20.9ns
4	8.4ns \times 6.0	12.5ns	13.7ns \times 2.0	21.2ns
5	10.3ns \times 19.8	14.8ns	25.4ns \times 1.9	21.3ns
6	8.0ns \times 78.6	13.4ns	29.5ns \times 7.4	21.5ns

classification. The corresponding results in Figure 10 indicate that Swift has a moderate performance advantage over LSF on all machines. For filter 3, compared to Opt-C, LSF takes a factor of 2.32 to 2.92 more time to process a packet, with an average slowdown of 2.67 times; Swift takes a factor of 1.22 to 2.07 more time to process a packet, with an average slowdown of 1.61 times. The average speedup of Swift over LSF is 1.43. For filter 4, compared to Opt-C, LSF takes a factor of 0.84 to 1.87 more time to process a packet, with an average slowdown of 1.48 times; Swift takes a factor of 0.65 to 0.95 more time to process a packet, with an average slowdown of 0.80 times. The average speedup of Swift over LSF is 1.39. Similar to the cases of filters 1 and 2, Table 6 shows that for filters 3 and 4, the per-instruction execution time of Swift is about 50% longer than that of LSF, but the much larger instruction count makes LSF slower than Swift in packet processing.

Filter 5 is a moderate load criterion designed to test the filtering engine’s capability of handling highly spe-

Table 7: Optimization effects of Swift filter 5

Filter 5	Original	Less-optimized	Unoptimized
Exec. Time	69.7ns	178.4ns	204.5ns

cific operation. The corresponding results in Figure 10 show that Swift outperforms LSF by a significant amount on all machines. Compared to Opt-C, LSF takes a factor of 3.68 to 6.38 more time to process a packet, with an average slowdown of 4.68 times; Swift takes a factor of 0.70 to 2.60 more time to process a packet, with an average slowdown of 1.39 times. The average speedup of Swift over LSF is 2.50. The significant speedup of Swift is due to its architectural advantages and specifically SIMD instructions. The ability to pack many operands (12 for TCP/DUP ports) in one instruction and batch the execution of comparison operations within a single filter engine “cycle” enables many-fold reduction at the cost of instruction interpretation, and improves the performance of Swift close to that of Opt-C. As shown in Table 6, compared to previous filters, LSF maintains its per-instruction execution time, but executes much more instructions. By contrast, Swift maintains its instruction count, and packs more operations in each instruction.

Filter 6 is a heavy load, “real life” criterion obtained from the campus network administrator. This filter is used by an application to detect suspicious IRC traffic. The filter is sufficiently complex for Swift to utilize the optimizations discussed earlier, namely SIMD instructions and hierarchical execution optimization. The corresponding results in Figure 10 show even higher performance increase of Swift against LSF. Compared to Opt-C, LSF takes a factor of 7.83 to 12.94 more time to process a packet, with an average slowdown of 10.09 times; Swift takes a factor of 1.79 to 5.21 more time to process a packet, with an average slowdown of 3.18 times. The average speedup of Swift over LSF is 2.79. According to Table 6, even though the per-instruction execution time of LSF is less than one third of Swift, the instruction count ratio between LSF and Swift is ten to one.

For all these filters, the auxiliary processing time for both LSF and Swift is fairly steady, as shown in Table 6. Although the auxiliary cost of Swift is about 8 to 9 nanoseconds more than that of LSF, the extra cost is insignificant as the filter becomes more complex and requires more time to execute.

We further measure the average execution time of Swift filter 5 with no SIMD extension (“Less-optimized”) and with neither SIMD extension nor hierarchical execution (“Unoptimized”) to provide more insights into the effect of Swift optimizations on performance improvement. The corresponding results are listed in Table 7. The removal of SIMD instructions exerts a great impact on the performance of the Swift filter, resulting in a slowdown of 156%. The comparatively small

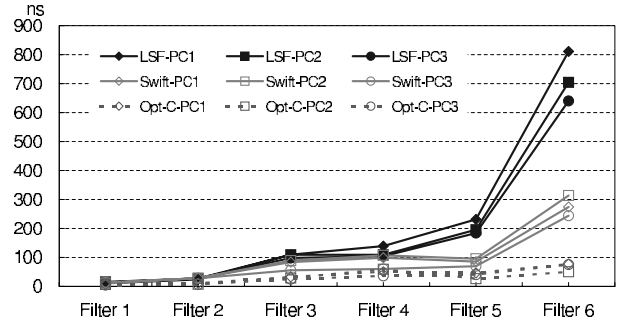


Figure 11: Per-packet processing time on all machines

increase of execution time after the removal of hierarchical execution indicates that the hierarchical execution has a minor effect on the performance of the Swift filter. For Swift filter 6, due to filter program organization, we remove hierarchical execution optimization for the “Less-optimized” experiment, and remove both hierarchical execution and SIMD extension for the “Unoptimized” experiment. Again, the impact of SIMD extension is far greater than that of hierarchical execution on the average execution time.

Overall, we find that (1) the SIMD extension plays a very important role in speeding up Swift filter execution, and (2) the hierarchical execution also helps the speedup but its effect is much smaller than that of SIMD extension, especially with a large instruction count. Without the SIMD extension and hierarchical execution, Swift can only perform comparably to optimized LSF for static filter tasks. These results are consistent with our observation from Figure 10 and Table 6: the speedup of Swift is mainly attributed to its use of much fewer instructions than LSF.

Figure 11 presents a comprehensive picture of filter execution time for LSF, Swift, and Opt-C filters among all machines categorized by six filtering criteria. It provides a good overview of the static filtering performance for cross comparison. When the filtering criteria are simple, LSF, Swift and Opt-C have nearly indistinguishable performance. As the criteria become more complex, the differences of filter execution time among the three filtering engines grow. Although both Swift and LSF run slower than Opt-C, the filter execution time of Swift grows at a much slower rate than that of LSF, and thus Swift achieves much closer performance to Opt-C than LSF.

6 Conclusion

This paper presents the design and implementation of the Swift packet filter. Swift provides an elegant, fast, and efficient packet filtering technique to handle the challenge of high speed network monitoring with dynamic filter updates. The key features of Swift lie in its low filter up-

date latency and high execution efficiency. Swift attains these performance advantages by embracing several major design innovations: (1) a specialized CISC instruction set increases filter execution efficiency and eliminates filter re-compilation, resulting in significantly reduced filter update latency; (2) a simple computational model removes the necessity of security checking and improves filter update latency; and (3) SIMD extensions further boost filter execution efficiency.

Our extensive experiments have validated Swift's efficacy and demonstrated the superiority of Swift against the *de facto* packet filter, BPF. For dynamic filtering tasks, the filter update latency of Swift is three orders of magnitude lower than that of BPF, and on each filter update, the number of packets missed by Swift is about two orders of magnitude less than that by BPF. For static filtering tasks, Swift runs as fast as BPF on simple filtering criteria, but is up to three times as fast as BPF on complex filtering criteria. Swift also performs much closer to optimized C filters than BPF.

There are many avenues we would like to further experiment and exploit in Swift. For instance, we will explore the multi-thread expansion of Swift, and develop a hardware optimized filter engine. We will make use of the extra registers supplied in the x86_64 processors for further performance improvement. Moreover, we envision that x86 high performance multimedia instructions (such as MMX and SSE) can also be used to accelerate the packet processing.

Acknowledgments

We are very grateful to our shepherd Eddie Kohler and the anonymous reviewers for their insightful and detailed comments, which have greatly improved the quality of the paper. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

References

- [1] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, affordable, high performance statistics collection. In *Proc. USENIX LISA'96*, pages 97–112, 1996.
- [2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proc. USENIX OSDI'94*, pages 115–123, 1994.
- [3] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proc. ACM SIGCOMM'99*, pages 123–134, 1999.
- [4] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly fast packet filters. In *Proc. USENIX OSDI'04*, pages 347–363, 2004.
- [5] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proc. of IEEE PAM'00*, pages 1–8, 2000.
- [6] J. Coppens, E. Markatos, J. Novotny, M. Polychronakis, V. Smotlacha, and S. Ubik. Scampi - a scaleable monitoring platform for the internet. In *Proc. 2nd Int'l Workshop on Inter-Domain Performance and Simulation*, 2004.
- [7] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. ACM CCS'04*, pages 2–11, 2004.
- [8] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM'96*, pages 53–59, 1996.
- [9] J. M. Gonzalez, V. Paxson, and N. Weaver. Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention. In *Proc. ACM CCS'07*, pages 139–149, 2007.
- [10] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: Packet filtering for low-cost network monitoring. In *Proc. IEEE HPSR'02*, pages 121–126, 2002.
- [11] V. Jacobson, C. Leres, and S. McCanne. Tcpdump(1). *Unix Manual Page*, 1990.
- [12] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *Proc. ACM/USENIX IMC 2005*, pages 267–272, 2005.
- [13] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proc. ACM SIGCOMM'98*, pages 215–227, 1998.
- [14] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX Technical Conference*, pages 259–269, 1993.
- [15] S. McCanne, C. Leres, and V. Jacobson. Libpcap. Available at <http://www.tcpdump.org/>. Lawrence Berkeley Laboratory, Berkeley, CA.
- [16] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. 11th ACM SOSP*, pages 39–51, 1987.
- [17] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. In *Proceedings of IEEE PAM'03*, 2003.
- [18] U. of Waikato. The DAG project. Available at <http://dag.cs.waikato.ac.nz/>.
- [19] C. Partridge, A. C. Snoeren, W. T. Strayer, B. Schwartz, M. Conde, and I. Castineyra. FIRE: Flexible intra-AS routing environment. In *Proc. SIGCOMM'00*, pages 191–203, 2000.
- [20] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.
- [21] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In *Proc. USENIX NSDI'04*, pages 141–153, 2004.
- [22] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proc. ACM FPGA'05*, pages 238 – 245, 2005.
- [23] A. Turner. Tcpreplay. <http://tcpreplay.synfin.net/trac/>.
- [24] J. van der Merwe, R. Caceres, Y. hua Chu, and C. Sreenan. mm-dump - a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.
- [25] G. Varghese. Network algorithmics - an interdisciplinary approach to designing fast networked devices. In *Morgan Kaufmann Publishers*, 2005.
- [26] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proc. 1994 Winter USENIX Technical Conference*, pages 153–165, 1994.

Securing Distributed Systems with Information Flow Control

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières
Stanford University

ABSTRACT

Recent operating systems [12, 21, 26] have shown that decentralized information flow control (DIFC) can secure applications built from mostly untrusted code. This paper extends DIFC to the network. We present DStar, a system that enforces the security requirements of mutually distrustful components through cryptography on the network and local OS protection mechanisms on each host. DStar does not require any fully-trusted processes or machines, and is carefully constructed to avoid covert channels inherent in its interface. We use DStar to build a three-tiered web server that mitigates the effects of untrustworthy applications and compromised machines.

1 INTRODUCTION

Software systems are plagued by security vulnerabilities in poorly-written application code. A particularly acute example is web applications, which are constructed for a specific web site and cannot benefit from the same level of peer review as more widely distributed software. Worse yet, a sizeable faction of web application code actually comes from third parties, in the form of libraries and utilities for tasks such as image conversion, data indexing, or manipulating PDF, XML, and other complex file formats. Indeed, faced with these kinds of tasks, most people start off searching for existing code. Sites such as FreshMeat and SourceForge make it easy to find and download a wide variety of freely available code, which unfortunately comes with no guarantee of correctness. Integrating third-party code is often the job of junior engineers, making it easy for useful but buggy third-party code to slip into production systems. It is no surprise we constantly hear of catastrophic security breaches that, for example, permit a rudimentary attacker to download 100,000 PayMaxx users' tax forms [20].

If we cannot improve the quality of software, an alternative is to design systems that remain secure despite untrustworthy code. Recent operating systems such as Asbestos [21], HiStar [26], and Flume [12] have shown this can be achieved through decentralized information flow control (DIFC). Consider again the PayMaxx example, which runs untrustworthy application code for each user to generate a tax form. In a DIFC operating system, we could sandwich each active user's tax form generation process between the payroll database and an HTTPS server, using the DIFC mechanism to prevent the application from communicating with any other component. Figure 1 illustrates this configuration. Suppose the

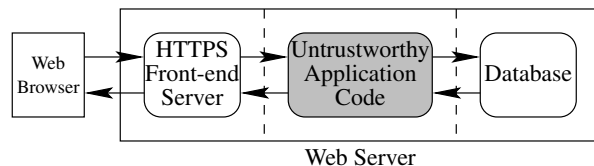


Figure 1: High-level structure of a web application. The shaded application code is typically the least trustworthy of all components, and should be treated as untrusted code to the extent possible. Dashed lines indicate how the web server can be decomposed into three kinds of machines for scalability—namely, front-end, application, and data servers.

HTTPS server forwards the username and password supplied by the browser to the database for verification, the database only allows the application to access records of the verified user, and the HTTPS server only sends the application's output to the same web browser that supplied the password. Then even a malicious application cannot inappropriately disclose users' data. Effectively we consider the application *untrusted*, mitigating the consequences if it turns out to be *untrustworthy*.

While DIFC OSes allow us to tolerate untrustworthy applications, they can do so only if all processes are running on the same machine. Production web sites, on the other hand, must scale to many servers. A site like PayMaxx might use different pools of machines for front-end HTTPS servers, application servers, and back-end database servers. To achieve Figure 1's configuration when each component runs on a separate machine, we also need a network protocol that supports DIFC.

This paper presents DStar, a protocol and framework that leverages OS-level protection on individual machines to provide DIFC in a distributed system. At a high level, DStar controls which messages sent *to* a machine can affect which messages sent *from* the machine, thereby letting us plumb together secure systems out of untrusted components. DStar was designed to meet the following goals:

Decentralized trust. While an operating system has the luxury of an entirely trusted kernel, a distributed system might not have any fully-trusted code on any machine. Web applications rely on fully-trusted certificate authorities such as Verisign to determine what servers to trust: for example, Google's web applications might trust any server whose certificate from Verisign gives it a name ending in *.google.com*. However, we believe that requiring such centralized trust by design needlessly stifles innovation and reduces security. Instead, DStar strives to provide a more general, decentralized mechanism that lets applications either use authorities such as Verisign

by convention or manage their trust in other ways. Moreover, the lack of a single trusted authority simplifies integration of systems in different administrative domains.

Egalitarian mechanisms. One of the features that differentiates DIFC work from previous information flow control operating systems is that DIFC mechanisms are specifically designed for use by applications. Any code, no matter how unprivileged, can still use DIFC to protect its data or further subdivide permissions. This property greatly facilitates interaction between mutually distrustful components, one of the keys to preserving security in the face of untrustworthy code. By contrast, military systems, such as [22], have also long controlled information flow, but using mechanisms only available to privileged administrators, not application programmers. Similarly, administrators can already control information flow in the network using firewalls and VLANs or VPNs. DStar's goal is to offer such control to applications and to provide much finer granularity, so that, in the PayMaxx example, every user's tax information can be individually tracked and protected as it flows through the network.

No inherent covert channels. Information flow control systems inevitably allow some communication in violation of policy through *covert channels*. However, different applications have different sensitivities to covert channel bandwidth. PayMaxx could easily tolerate a 1,000-bit/second covert channel: even if a single instance of the untrusted application could read the entire database, it would need almost a day to leak 100,000 user records of 100 bytes each. (Exploiting covert channels often involves monopolizing resources such as the CPU or network, which over such a long period could easily be detected.) By contrast, military systems consider even 100 bits/second to be high bandwidth. To ensure that DStar can adapt to different security requirements, our goal is to avoid any covert channels inherent in its interface. This allows any covert channels to be mitigated in a particular deployment without breaking backwards compatibility. Avoiding inherent covert channels is particularly tricky with decentralized trust, as even seemingly innocuous actions such as querying a server for authorization or allocating memory to hold a certificate can inadvertently leak information.

DStar runs on a number of operating systems, including HiStar, Flume, and Linux. DStar leverages the OS's DIFC on the former two, but must trust all software on Linux. Nonetheless, running DStar on Linux is convenient for incremental deployment; for example we can run the least-trusted application components on HiStar or Flume, and sandwich them between existing Linux web server and database machines.

To illustrate how DStar is used, Section 5.1 describes a secure web server we built on HiStar, and Section 5.3 shows how we distributed it using DStar. On a single

machine, our web server ensures that the SSL certificate private key is accessible only to a small part of the SSL library, and can only be used for legitimate SSL negotiation. It also protects authentication tokens, such as passwords, so that even the bulk of the authentication code cannot disclose them. In a distributed setting, DStar allows the web server to ensure that private user data returned from a data server can only be written to an SSL connection over which the appropriate user has authenticated himself. DStar's decentralized model also ensures that even web server machines are minimally trusted: if any one machine is compromised, it can only subvert the security of users that use or had recently used it.

2 INFORMATION FLOW CONTROL

DStar enforces DIFC with labels. This section first describes how these labels—which are similar to those of DIFC OSes—can help secure a web application like PayMaxx. We then detail DStar's specific label mechanism.

2.1 Labels

DStar's job is to control how information flows between processes on different machines—in other words, to ensure that only processes that should communicate can do so. Communication permissions are specified with *labels*. Each process has a label; whether and in which direction two processes may communicate is a function of their labels. This function is actually a partial order, which we write \sqsubseteq (pronounced “can flow to”).

Roughly speaking, given processes S and R labeled L_S and L_R , respectively, a message can flow from S to R only if $L_S \sqsubseteq L_R$. Bidirectional communication is permitted only if, in addition, $L_R \sqsubseteq L_S$. Labels can also be incomparable; if $L_S \not\sqsubseteq L_R$ and $L_R \not\sqsubseteq L_S$, then S and R may not communicate in either direction. Such label mechanisms are often referred to as “no read up, no write down,” where “up” is the right hand side of \sqsubseteq . Given that labels can be incomparable, however, a more accurate description might be, “only read down, only write up.”

Because a distributed system cannot simultaneously observe the labels of processes on different machines, DStar also labels messages. When S sends a message M to R , S specifies a label L_M for the message. The property enforced by DStar is that $L_S \sqsubseteq L_M \sqsubseteq L_R$. Intuitively, the message label ensures that untrusted code cannot inappropriately read or disclose data. In the PayMaxx example, the payroll database should use a different value of L_M to protect the data of each user in the database, so that only the appropriate instance of the tax form generating application and HTTPS front ends can receive messages containing a particular user's data.

2.2 Downgrading privileges

If data could flow only to higher and higher labels, there would be no way to get anything out of the system: the

high label of the HTTPS front end would prevent it from sending tax forms back to a client web browser over a network device with a low label. The big difference between DIFC and more traditional information flow [8] is that DIFC *decentralizes* the privilege of bypassing “can flow to” restrictions. Each process P has a set of privileges O_P allowing it to omit particular restrictions when sending or receiving messages, but no process has blanket permission to do so. In effect, O_P lets P *downgrade* or lower labels on message data in certain ways.

We write \sqsubseteq_{O_P} (“can flow to, given privileges O_P ”) to compare labels in light of a set of privileges O_P . As we will show later, \sqsubseteq_{O_P} is strictly more permissive than \sqsubseteq ; in other words, $L_1 \sqsubseteq L_2$ always implies $L_1 \sqsubseteq_{O_P} L_2$, but not vice versa. What DStar actually enforces, then, is that S can send a message M that R receives only if

$$L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R.$$

In other words, S with privileges O_S can produce a message labeled L_M and R with privileges O_R can receive a message so labeled. In the PayMaxx example, one might give the untrusted tax form generator no privileges, while giving the HTTPS front end the ability to downgrade L_M so as to send data back to the client web browser.

But who assigns such labels and privileges to the different processes in a system? Very often the security policy we want is a conjunction of concerns by mutually distrustful components. For example, the payroll database may want to disclose user records only after the the front-end server has supplied the appropriate password, while the front-end server may want to prevent the database from disclosing plaintext passwords sent to it for authentication. If DStar concentrated the ability to set labels in a group of specially-designated, privileged processes, it would impede the ability of unprivileged software to express security requirements. That, in turn, would lead to more trusted components and increased damage should any of them in fact be untrustworthy. Fortunately, as described in the next subsection, decentralized downgrading helps make DStar’s labels an egalitarian mechanism with which any process can express security concerns.

2.3 Categories

A DStar label is a set of *categories*, each of which imposes a restriction on who can send or receive data. There are two types of category: *secrecy* and *integrity*. Secrecy categories in a message label restrict who can receive the message. In the PayMaxx example, to avoid improper disclosure, the database server should have one secrecy category for each user and include that category in the label of any messages containing that user’s data. Conversely, integrity categories in a message label constrain who may have sent the message, and thus can help authenticate the sender. We will use s and i subscripts to indicate secrecy and integrity categories, respectively.

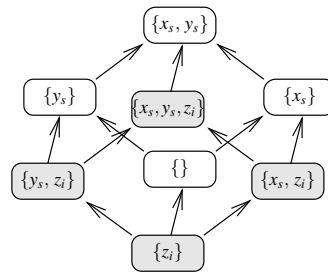


Figure 2: Lattice formed by labels using two secrecy categories, x_s and y_s , and an integrity category, z_i . Shading indicates labels that include the z_i category. Arrows show pairs of labels where the “can flow to” \sqsubseteq relation holds, and thus how messages can be sent. Self-arrows are not shown. Information can also flow transitively over multiple arrows.

For any two labels L_1 and L_2 , we can now formally define the \sqsubseteq relation:

$L_1 \sqsubseteq L_2$ if and only if L_1 contains all the integrity categories in L_2 and L_2 contains all the secrecy categories in L_1 .

As illustrated in Figure 2, labels form a lattice under the \sqsubseteq relation, and thus transitively enforce a form of mandatory access control [8].

A process P ’s downgrading privileges, O_P , are also represented as a set of categories. We say P *owns* a category c when $c \in O_P$. Ownership confers the ability to ignore the restrictions imposed by a particular category at the owner’s discretion. For any two labels L_1 and L_2 and privileges O , we can now formally define \sqsubseteq_O :

$L_1 \sqsubseteq_O L_2$ if and only if $L_1 - O \sqsubseteq L_2 - O$.

In other words, except for the categories in O , L_1 contains all the integrity categories in L_2 and L_2 contains all the secrecy categories in L_1 .

What makes categories egalitarian is that any process can allocate a category and simultaneously gain ownership of the newly allocated category. A process that owns a category may also, at its discretion, *grant* ownership of that category to another process. Every DStar message M includes a set of categories G_M that its sender S is granting to the recipient, where DStar ensures that

$$G_M \subseteq O_S.$$

Thus, the payroll database can create one category per user and grant ownership of the category to the appropriate HTTPS front end, as illustrated in Figure 3. At the same time, the front end can allocate its own categories to protect plaintext passwords sent to the database.

2.4 Clearance

In addition to its label L_P and privileges O_P , each process P has a third set of categories, C_P , called its *clearance*. Clearance represents the right of a process to raise its own label. A process may set its label to any value L_P^{new}

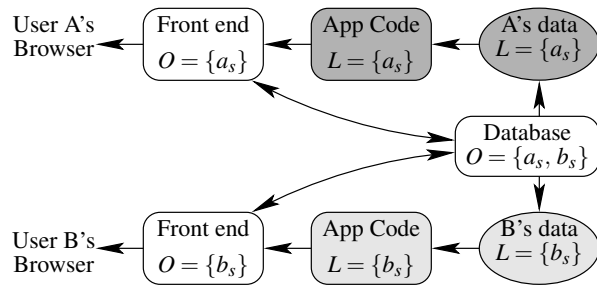


Figure 3: Example use of labels to prevent the PayMaxx application code from inappropriately disclosing data, with two users, A and B. Rounded boxes are processes. Ellipses are messages. Shaded components are labeled with a user's secrecy category, a_s or b_s for A and B respectively. The front end communicates with the database to authenticate the user and obtain ownership of the user's secrecy category.

such that $L_P \subseteq L_P^{\text{new}} \subseteq C_P$. A process that owns a category can raise the clearance of other processes in that category. For instance, when the front-end server sends a user's password to the payroll database, the database responds by granting it ownership of the user's secrecy category. Given this ownership, the front end can raise the clearance of the application, which in turn allows the application to add that user's secrecy category to its label. DStar fully implements clearance, but for simplicity of exposition we mostly omit it from discussion.

3 DSTAR EXPORTER

To model the fact that all processes with direct access to a particular network can exchange messages, DStar requires all such processes to have the same label, L_{net} , which we call the *network label*. (For networks connected to the Internet, $L_{\text{net}} = \{\}$.) However, our main goal of restricting the communication of potentially untrustworthy code requires running different processes with different labels. In order to allow these processes, which lack direct network access, to communicate across machines, DStar introduces the notion of *exporter* daemons.

Each host runs an exporter daemon, which is the only process that sends and receives DStar messages directly over the network. When two processes on different machines communicate over DStar, they exchange messages through their local exporters, as shown in Figure 4. Exporters are responsible for enforcing the information flow restrictions implied by message labels, which boils down to three requirements. Each exporter E must:

1. Track the labels assigned to, and the categories owned by, every process on E 's machine,
2. Ensure that processes on E 's machine cannot send or receive messages with inappropriate labels, and
3. Send or accept a network message M only when E can trust the remote exporter to respect the communication restrictions implied by M 's label.

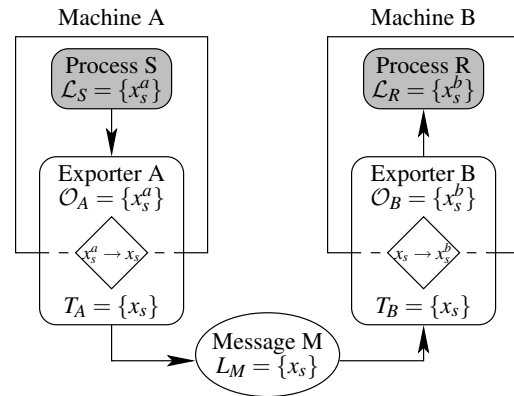


Figure 4: Message sent by process S on machine A to process R on machine B . \mathcal{L} and \mathcal{O} designate local OS labels and OS ownership privileges. x_s^a and x_s^b are local OS secrecy categories. Exporters translate between these OS categories and the globally-meaningful DStar category x_s . T_A is the set of DStar categories whose owners explicitly trust exporter A, while T_B is the analogous set for exporter B.

The next three subsections discuss each of these requirements in turn.

3.1 Local OS DIFC

To track the labels and ownership privileges of local processes, exporters rely on the DIFC facilities of the local operating system. Asbestos, HiStar, and Flume all have mechanisms similar to DStar's categories and labels. We will always refer to these as *OS categories* and *OS labels* to differentiate them from DStar's categories and labels. Every exporter establishes mappings between OS categories, which have meaning only on the local machine, and DStar categories, which are globally meaningful. At a high level, one can view the exporter's job as translating between local OS DIFC protection of data on the host and cryptographic protection of data on the network. Because an exporter's label is always the network label, the exporter must own all local OS categories required to access the messages it processes.

Not every OS category has an equivalent DStar category. However, every category in a process P 's DStar label, L_P , is represented by a corresponding OS category in P 's OS label, \mathcal{L}_P . Each restriction implied by a DStar category is thus locally enforced by the corresponding OS category, thereby ensuring processes on the same machine cannot use local OS facilities to communicate in violation of their DStar labels. Similarly, every DStar category that P owns is reflected in P 's local OS downgrading privileges, \mathcal{O}_P . Processes must explicitly invoke the exporter to create a mapping between a local and a DStar category. Section 4.2 details this process for HiStar. An important point is that the exporter creates unforgeable mappings, but does not store them. Processes must arrange to store the mappings they care about and specify them when sending messages.

3.2 Local exporter checks

Recall from Section 2.2 that when a process S sends a message M to a process R on another machine, DStar must ensure $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R$. If M grants ownership privileges, DStar must also ensure $G_M \subseteq O_S$.

The left half of the first requirement, $L_S \sqsubseteq_{O_S} L_M$, and the second requirement, $G_M \subseteq O_S$, are enforced on S 's machine. Roughly speaking, the local OS ensures that these relations hold in terms of local categories, and the exporter translates local categories into DStar categories. In particular, when S sends M to S 's exporter through the local OS's IPC mechanism, it uses the OS's DIFC both to label the message \mathcal{L}_M (requiring $\mathcal{L}_S \sqsubseteq_{O_S} \mathcal{L}_M$) and to prove to the exporter that it owns \mathcal{G}_M (requiring $\mathcal{G}_M \subseteq O_S$). S 's exporter, in turn, uses mappings it previously created to translate the OS categories in \mathcal{L}_M and \mathcal{G}_M into DStar categories to produce L_M and G_M .

The right half of DStar's first requirement, $L_M \sqsubseteq_{O_R} L_R$, is enforced on R 's machine in an analogous fashion. Upon receiving M from the network, R 's exporter translates the DStar categories in L_M and G_M into OS categories to obtain a local OS message label \mathcal{L}_M and set of OS categories \mathcal{G}_M that should be granted to R . (It performs this translation using mappings it previously created, which must be explicitly referenced in the message.) The exporter then sends M through the local OS's IPC mechanism to R with a label of \mathcal{L}_M , and grants R ownership of categories in \mathcal{G}_M . The local OS DIFC mechanism, in turn, ensures that $\mathcal{L}_M \sqsubseteq_{O_R} \mathcal{L}_R$.

Together, the two exporters guarantee $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R$, but only if they are both trustworthy. The final task of an exporter is to determine whether or not it can trust the remote exporter to enforce its half of the equation.

3.3 Decentralized trust

When should an exporter trust another exporter to send or receive a message M ? In order to support decentralized trust, DStar leaves this decision up to the owners of the categories whose security is at stake—namely those in which L_M differs from L_{net} , and those in G_M . Each exporter E has a set of categories T_E whose owners trust E to handle the category. We refer to T_E as E 's *trust set*.

When a process P creates a DStar category c , P implicitly adds c to its local exporter's trust set. Before P can either grant ownership of c to a remote process Q or raise Q 's clearance in c , P must explicitly add c to the trust set of Q 's exporter. This reflects the reality that a process can only be as trustworthy as its exporter.

Adding category c to an exporter E 's trust set has the same security implications as granting ownership of c to a process. The difference is that while an exporter has no way of verifying the local OS ownership privileges of a process on another machine, it *can* verify the trust set of a remote exporter.

Consider S on machine A sending M to R on machine B , as in Figure 4. Treating trust as ownership, we can view this as two message exchanges. From exporter A 's point of view, M flows from S to exporter B . Therefore, DStar should ensure that $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{T_B} L_B$ and $G_M \subseteq O_S$. From exporter B 's point of view, the message flows from exporter A to R , which requires $L_A \sqsubseteq_{T_A} L_M \sqsubseteq_{O_R} L_R$ and $G_M \subseteq T_A$. Because they directly access the network, both exporters have the same label, $L_A = L_B = L_{\text{net}}$. It therefore remains to verify each other's trust sets.

DStar implements trust sets using certificates. Each exporter has a public/private key pair. If an exporter is trusted in a particular category, it can use its private key to sign a certificate delegating trust in that category to another exporter, named by its public key. Certificates include an expiration time to simplify revocation.

To avoid any trusted, central naming authority, DStar uses *self-certifying* category names that include the public key of the exporter that created the category. An exporter is trusted in every category it creates by definition. Given a category's name, other exporters can verify certificates signed by its creator using the public key in the name. Further delegation of trust requires the entire chain of certificates leading to the category's creator.

In many cases, even contacting another machine over the network to query it for credentials can inappropriately leak information. Thus, an important property of DStar certificates is that they allow one exporter to verify membership of a category in a remote exporter's trust set with no external communication.

3.4 Addressing

Delegation certificates determine when an exporter with a particular public key can receive a message with a particular message label. However, at a low level, network messages are sent to network addresses, not public keys. Any communication to map exporter public keys to network addresses could in itself leak information. We therefore introduce *address certificates*, which contain the exporter's current IP address signed by the exporter's key. Exporters only send messages to other exporters for which they have address certificates.

Unfortunately, most networks cannot guarantee that packets sent to an IP address will only reach the intended recipient. On a typical Ethernet, an active attacker can spoof ARP replies, overflow MAC address tables, or flood the network to gain information about communication patterns between other hosts. While DStar encrypts and MACs its network traffic, the mere presence and destination of encrypted packets may communicate information. Malicious code on a HiStar machine could, for instance, exploit this fact to leak information to a colluding Linux box on the same network. While the present level of trust in the network suffices for many applications,

in the future we intend to integrate DStar with network switches that can better conceal communication [6].

Exporters currently distribute address certificates by periodically broadcasting them to the local-area network. Certificate expiration times allow IP address reuse. After expiration, other exporters will not connect to the old address. In a complex network, broadcast would not suffice to distribute address certificates to all exporters; one might need a partially-trusted directory service.

3.5 Exporter interface

Exporters provide unreliable one-way message delivery to communication endpoints called *slots*, which are analogous to message ports. Communication is unreliable to avoid potential covert channels through message acknowledgments; for instance, if process *S* can send messages to *R* but not vice-versa, a message delivery status could allow *R* to convey information to *S* by either accepting or refusing messages. However, in the common case where labels permit bi-directional communication, library code outside the exporter provides higher-level abstractions such as RPC, much the way RPC can be layered on top of unreliable transports such as UDP and IP.

We will now describe the DStar network protocol and exporter interface, starting with DStar's self-certifying category names:

```
struct category_name {
    pubkey creator;
    category_type type;
    uint64_t id;
};
```

Here, *type* specifies whether this is a secrecy or integrity category, and *id* is an opaque identifier used to distinguish multiple categories created by the same exporter. Exporters can create categories by picking a previously unused pseudo-random *id* value, for example by encrypting a counter with a block cipher.

Next, the format of all DStar messages is as follows:

```
struct dstar_message {
    pubkey recipient_exporter;
    slot recipient_slot;
    category_set label, ownership, clearance;
    cert_set certs;
    mapping_set mapset;
    opaque payload;
};
```

The message is addressed to slot *recipient_slot* on *recipient_exporter*'s machine. The *label* specifies information flow restrictions on the contents of the message, and consists of a set of *category_names* as defined earlier. The recipient exporter will grant ownership and clearance of categories specified in *ownership* and *clearance* respectively to the recipient slot when it delivers the message.

certs contains delegation certificates proving to the recipient exporter that the sender is trusted with all of the categories in *ownership* and *clearance* (stated as $G_M \subseteq T_{\text{sender}}$ in Section 3.3), and, assuming $L_{\text{sender}} = L_{\text{net}} = \{\}$, trusted with all *integrity* categories in *label* (stated as $L_{\text{sender}} \sqsubseteq_{T_{\text{sender}}} L_M$ in Section 3.3). *mapset* contains mappings for the recipient exporter to map the necessary DStar categories to OS categories; we discuss these mappings in more detail in Section 4.2.

Each exporter provides to other processes on the same machine a single function to send DStar messages:

```
void dstar_send(ip_addr, tcp_port, dstar_message,
               cert_set, mapping_set);
```

Here, the *cert_set* and *mapping_set* have the opposite roles from those in *dstar_message*. They prove to the *sending* exporter that it is safe to send the supplied *dstar_message* to the recipient exporter. In particular, assuming $L_{\text{recipient}} = L_{\text{net}} = \{\}$, *cert_set* contains delegation certificates proving the recipient exporter is trusted in all *secrecy* categories in the message label (stated as $L_M \sqsubseteq_{L_{\text{recipient}}} L_{\text{recipient}}$ in Section 3.3), while *mapping_set* provides mappings allowing the sending exporter translate the necessary OS categories to DStar categories. *cert_set* must also include an address certificate proving that the given IP address and TCP port number reach the recipient exporter.

Finally, delivery of messages by an exporter to local processes will be discussed in Section 4.3.

3.6 Management services

DStar exporters provide additional functionality for management and bootstrapping, implemented as RPC servers on well-known slots. We will later illustrate how they are used in an application.

The **delegation service** allows a process that owns a category in the local operating system to delegate trust of the corresponding DStar category to another exporter, named by a public key. A signed delegation certificate is returned to the caller. This service is fully trusted; a compromise would allow an attacker to create arbitrary delegations and gain full control over all data handled by an exporter.

The **mapping service** creates mappings between DStar categories and local operating system security mechanisms; it will be discussed in more detail in Section 4.2. This service is also fully trusted.

The **guarded invocation service** launches executables with specified arguments and privileges, as long as a cryptographic checksum of the executable matches the checksum provided by the caller. The caller must have access to memory and CPU resources on the remote machine in order to run a process. This service is used in bootstrapping, when only the public key of a trusted exporter is known; the full bootstrapping process will be

described in more detail later on. This service is not trusted by the exporter—in other words, its compromise cannot violate the exporter’s security guarantees. However, the exporter reserves a special slot name for this service, so that clients can contact this service on a remote machine during bootstrapping.

Finally, exporters provide a **resource allocation service**, which allows allocating resources, such as space for data and CPU time for threads, with a specified label on a remote machine. This service is used by the client-side RPC library to provide the server with space for handling the request; a timeout can be specified for each allocation, which allows garbage-collecting ephemeral RPC call state in case of a timeout. In the PayMaxx example, the front-end server uses this service to allocate memory and CPU time for running the application code on the application server. The system administrator also uses this service to bootstrap new DStar machines into an existing pool of servers, as will be described in Section 5.6. Although we describe this service here because it is used by a number of applications, it is not strictly necessary for an exporter to provide this service: applications could invoke the resource allocation service by using guarded invocation.

4 HiSTAR EXPORTER

DStar exporters use the local operating system to provide security guarantees, and this section describes how exporters use HiStar and its category and label mechanism for this purpose. DStar also runs on Flume; the main difference there is that Flume does not explicitly label all memory resources, which can lead to covert channels or denial-of-service attacks.

To reduce the effect of any compromise, the HiStar exporter has *no superuser privileges*. The exporter runs as an ordinary process on HiStar without any special privileges from the kernel. The owner of each local category can explicitly allow the exporter to translate between that category on the local machine and encrypted DStar messages, by granting ownership of the local HiStar category to the exporter. The exporter uses this ownership to allow threads labeled with that category, which may not be able to send or receive network messages directly, to send and receive appropriately-labeled DStar messages.

To avoid covert channels, the HiStar exporter is largely stateless, keeping no per-message or per-category state. Instead, the exporter requires users to explicitly provide any state needed to process each message.

4.1 HiStar review

The HiStar kernel interface is designed around a small number of object types, including *segments*, *address spaces*, *containers*, *threads*, and *gates*. Like DStar, HiStar uses the notion of categories (implemented as

opaque 61-bit values in the kernel) to specify information flow restrictions, and each object has a unique 61-bit object ID, and a label used for access control by the kernel, similar to a DStar label. For simplicity, this paper will use DStar labels to describe the labels of HiStar objects, although in reality an equivalent HiStar label is used, in which secrecy categories map to level **3**, integrity categories map to level **0**, ownership of categories maps to level *****, and the default level is **1**.

The simplest object type, a *segment*, consists of zero or more memory pages. An *address space* consists of a set of $VirtualAddress \rightarrow SegmentID$ mappings that define the layout of a virtual address space.

Containers are similar to directories, and all objects must exist in some container to avoid garbage collection. The root container is the only object in the system that does not have a parent container. Containers provide all resources in HiStar, including storage (both in memory and on disk) and CPU time.

Threads execute code, and consist of a register set and an object ID of an address space object that defines the virtual address space. Threads also have privileges—a set of categories owned by the thread. Any thread can allocate a fresh category, at which point it becomes the only one in the system with ownership of the new category.

Gates are used for IPC and privilege transfer. Unlike a typical IPC message port, gates require the client to donate initial resources—the thread object—for execution in the server’s address space. Like threads, gates have privileges, which can be used when the thread switches to the server’s address space. Each gate has an *access label* that controls who can invoke the gate: a thread T can invoke gate G with access label A_G only if $L_T \sqsubseteq_{O_T} A_G$.

4.2 Category mappings

One of the main tasks of the exporter is to translate between the global space of DStar categories and the corresponding HiStar categories on the local machine. Since the exporter must be stateless, it is up to the users to supply these mappings for each message. However, these mappings are crucial to the security guarantees provided by the exporter—by corrupting these mappings, an attacker could convince the exporter to label an incoming secret message with a category owned by the attacker on the local machine, violating all security guarantees.

In the network protocol, exporters use signed certificates to get around this problem: users supply certificates to send each message, but exporters verify the signature on each certificate. However, on the local machine exporters also need ownership of the local HiStar category in order to be able to manipulate data labeled with that category. Since the HiStar kernel only allows category ownership to be stored in thread or gate objects, the ex-

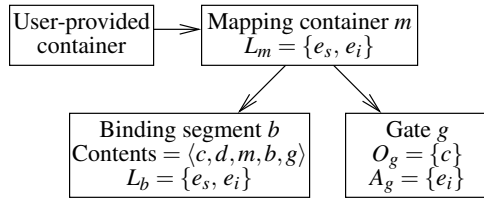


Figure 5: Objects comprising a mapping between DStar category d and local HiStar category c . Arrows indicate that an object is in a container.

porter fundamentally requires memory (for a kernel object) for each category it handles on the local machine.

Thus, for each mapping between a DStar category and a HiStar category, the exporter needs two things: a kernel object storing ownership of the local HiStar category, and a secure binding between the DStar and HiStar category names. The secure binding could be represented by a certificate, but since the exporter already needs a kernel object to store ownership, we store the secure binding along with that kernel object, and avoid the overhead of public key cryptography.

HiStar’s exporter represents each mapping using the objects shown in Figure 5. Container m stores all other mapping objects, and in turn lives in a user-provided container, which allows the exporter itself to remain stateless. Gate g stores the exporter’s ownership of the local HiStar category. Finally, binding segment b ensures that the user cannot tamper with the mapping despite providing the memory for the kernel objects, as follows.

The exporter owns two HiStar categories, e_s and e_i , which it uses to ensure the security of *all* mappings on its machine. All objects comprising a mapping are labeled $\{e_s, e_i\}$, which ensures that only the exporter can create or modify them. The binding segment provides a secure binding between the DStar and HiStar category names, and contains the *mapping tuple*, $\langle c, d, m, b, g \rangle$. Users provide this tuple when they want to use a particular category mapping; the *mapping_set*, mentioned earlier in the *dstar_message* and the *dstar_send()* function, is a set of such tuples. The exporter verifies each tuple’s integrity by checking that the tuple matches the contents of the binding segment, and that the binding segment has label $\{e_i, e_s\}$, so that only the exporter could have created it. Finally, to ensure that only the exporter can gain ownership of the HiStar category through the mapping gate, the gate’s access label is set to $A_g = \{e_i\}$.

The mapping service, briefly mentioned earlier, allows applications to create new mappings. This service allows anyone to allocate either a fresh HiStar category for an existing DStar category, or a fresh DStar category for an existing HiStar category. Since the exporter is stateless, the caller must provide a container to store the new mapping, and grant the mapping service any privileges needed to access this container. The exporter does not

grant ownership of the freshly-allocated category to the caller, making it safe for anyone to create fresh mappings. If the calling process does not own the existing category, it will not own the new category either, and will not be able to change the security policy set by the owner of the existing category. If the calling process does own the existing category, it can separately gain ownership of the new category, by sending a message that includes the existing category in the ownership field.

The mapping service also allows creating a mapping between an existing pair of HiStar and DStar categories, which requires the caller to prove ownership of both categories, by granting them to the mapping service.

4.3 Exporter interface

Exporters running on HiStar support two ways of communicating with other processes running on the same machine: *segments* and *gates*. Communicating via a segment resembles shared memory: it involves writing the message to the segment and using a futex [10] to wake up processes waiting for a message in that segment. Communication over a gate involves writing the message to a new segment, and then allocating a new thread, which in turn invokes the gate, passing the object ID of the message segment. Gates incur higher overhead for sending a message than segments, but allow passing ownership and clearance privileges when the thread invokes the gate.

As mentioned earlier, messages are delivered to slots, which in the case of HiStar names either a segment (by its object ID), or a gate (by its object ID and the object ID of a container to hold the newly-created message segment and thread). Exporters enforce labels of incoming messages by translating DStar labels into local HiStar labels, and making sure that the label of the slot matches the label of the message. Similarly, when a process sends a message, exporters ensure that the message label is between the label and clearance of the sending process.

The local exporter saves all address certificates it receives via broadcast from other exporters to a well-known file. This makes it easy for other processes to find address certificates for nearby exporters.

4.4 Implementation

The exporter comprises about 3,700 lines of C++ source code, and runs on HiStar, Flume, and Linux (though on Linux, all software must be trusted to obey information flow restrictions). The client library, trusted by individual processes to talk to the exporter, is 1,500 lines of C and C++ code. The exporter uses the libsync event-driven library [13] for network I/O and cryptography, and libc and libstdc++, which dwarf it in terms of code size.

5 APPLICATIONS

To illustrate how DStar helps build secure distributed systems, we focus on two scenarios that expand on our

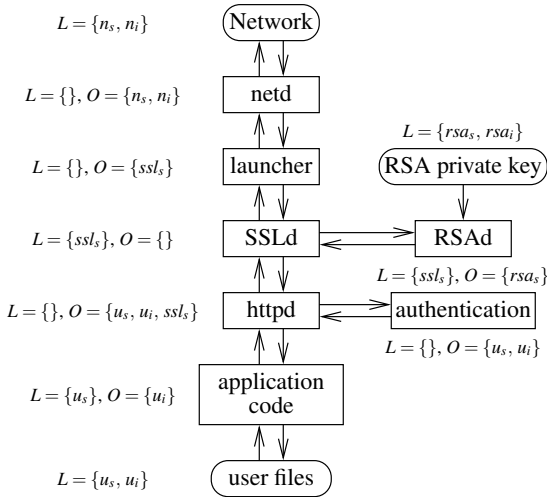


Figure 6: Architecture of the HiStar SSL web server. Rectangles represent processes. Rounded boxes represent devices and files.

running PayMaxx example. First, we describe the architecture of a highly privilege-separated web server we have built on HiStar, which partitions its privileges among many separate components to limit the effect of any single compromise, and discuss the security properties it achieves on one machine. We then show how this web server can be distributed over multiple machines like a typical three-tiered web application, providing performance scalability with similar security guarantees as on a single HiStar machine. Finally, we show how, even in an existing web service environment, DStar can be used to improve security by incrementally adding information flow control for untrusted code.

5.1 Web server on HiStar

Figure 6 shows the overall architecture of our privilege-separated SSL web server. The web server is built from a number of mutually-distrustful components to reduce the effects of the compromise of any single component. We first describe how requests are handled in this web server, and the next subsection will describe its security.

The TCP/IP stack in HiStar is implemented by a user-space process called *netd*, which has direct access to the kernel network device. *netd* provides a traditional sockets interface to other applications on the system, which is used by our web server to access the network.

User connections are initially handled by the *launcher*, which accepts incoming connections from web browsers and starts the necessary processes to handle them. For each request, the *launcher* spawns *SSLd* to handle the SSL connection with the user's web browser, and *httpd* to process the user's plaintext HTTP request. The *launcher* then proxies data between *SSLd* and the TCP connection to the user's browser. *SSLd*, in turn, uses the *RSAd* daemon to establish an SSL session key with

the user's web browser, by generating an RSA signature using the SSL certificate private key kept by *RSAd*.

httpd receives the user's decrypted HTTP request from *SSLd* and extracts the user's password and request path from it. It then authenticates the user, by sending the user's password to that user's *password checking agent* from the HiStar authentication service [26]. If the authentication succeeds, *httpd* receives ownership of the user's secrecy and integrity categories, u_s and u_i , and executes the *application code* with the user's privileges (in our case, we run GNU ghostscript to generate a PDF document). Application output is sent by *httpd* back to the user's web browser, via *SSLd* for encryption.

5.2 Web server security

The HiStar web server architecture has no hierarchy of privileges, and no fully trusted components; instead, most components are mutually distrustful, and the effects of a compromise are typically limited to one user, usually the attacker himself. Figure 7 summarizes the security properties of this web server, including the complexity of different components and effects of compromise.

The largest components in the web server, *SSLd* and the application code, are minimally trusted, and cannot disclose one user's private data to another user, even if they are malicious. The application code is confined by the user's secrecy category, u_s , and it is *httpd*'s job to ensure that the application code is labeled with u_s when *httpd* runs it. Although the application code owns the user's integrity category, u_i , this only gives it the privilege to write to that user's files, and not to export them. Ownership of u_i is necessary to allow the application code to read data not labeled with u_i , such as shared libraries. If the application code were to be labeled with u_i instead, it would be restricted to reading only data labeled u_i , which would likely exclude needed binaries, shared libraries, and configuration files.

SSLd is confined by ssl_s , a fresh secrecy category allocated by the *launcher* for each new connection. Both the *launcher* and *httpd* own ssl_s , allowing them to freely handle encrypted and decrypted SSL data, respectively. However, *SSLd* can only communicate with *httpd* and, via the *launcher*, with the user's web browser.

SSLd is also not trusted to handle the SSL certificate private key. Instead, a separate and much smaller daemon, *RSAd*, has access to the private key, and only provides an interface to generate RSA signatures for SSL session key establishment. Not shown in the diagram is a category owned by *SSLd* that allows it and only it to invoke *RSAd*. Although a compromised *RSAd* can expose the server's SSL private key, it cannot directly compromise the privacy of user data, because *RSAd* runs confined with each user connection's ssl_s category.

Component	Lines of Code	Label	Ownership	Effects of Compromise
netd	350,000	{}	$\{n_s, n_i\}$	Equivalent to an active network attacker; subject to same kernel label checks as any other process
launcher	310	{}	$\{ssl_s\}$	Obtain plaintext requests, including passwords, and subsequently corrupt user data
SSLd	340,000	$\{ssl_s\}$	{}	Corrupt request or response, or send unencrypted data to same user's browser
RSAd	4,600	$\{ssl_s\}$	$\{rsa_s\}$	Disclose the server's SSL certificate private key
htpd	300	{}	$\{u_s, u_i, ssl_s\}$	Full access to data in attacker's account, but not to other users' data
authentication	320	{}	$\{u_s, u_i\}$	Full access to data of the user whose agent is compromised, but no password disclosure
application	680,000+	$\{u_s\}$	$\{u_i\}$	Send garbage (but only to same user's browser), corrupt user data (for write requests)
DStar exporter	3,700			Corrupt or disclose any data sent or received via DStar on a machine
DStar client library	1,500			Corrupt, but not necessarily disclose, data sent or received via DStar by an application

Figure 7: Components of the HiStar web server, their complexity measured in lines of C code (not including libraries such as libc), their label and ownership, and the worst-case results of the component being compromised. The netd TCP/IP stack is a modified Linux kernel; HiStar also supports the lwIP TCP/IP stack, consisting of 35,000 lines of code, which has lower performance. The DStar exporter and client library illustrate the additional code that must be trusted in order to distribute this web server over multiple machines.

Side-channel attacks, such as [1], might allow recovery of the private key; OpenSSL uses RSA blinding to defeat timing attacks such as [5]. To prevent an attacker from observing intermediate states of CPU caches while handling the private key, *RSAd* starts RSA operations at the beginning of a 10 msec scheduler quantum (each 1024-bit RSA operation takes 1 msec), and flushes CPU caches when context switching to or from *RSAd* (with kernel support), at a minimal cost to overall performance.

The HiStar authentication service used by *htpd* to authenticate users is described in detail in [26], but briefly, there is no code executing with every user's privilege, and the supplied password cannot be leaked even if the password checker is malicious.

In our current prototype, *htpd* always grants ownership of u_i to the application code, giving it write access to user data. It may be better to grant u_i only to code that performs read-write requests, to avoid user data corruption by buggy read-only request handling code.

Our web server does not use SSL client authentication in *SSLd*. Doing so would require either trusting all of *SSLd* to authenticate all users, or extracting the client authentication code into a separate, smaller trusted component. In comparison, the password checking agent in the HiStar authentication service is 320 lines of code.

One caveat of our prototype is its lack of SSL session caching. Because a separate instance of *SSLd* is used for each client request, clients cannot reuse existing session keys when connecting multiple times, requiring public key cryptography to establish a new session key. This limitation can be addressed by adding a trusted SSL session cache that runs in a different, persistent process, at the cost of increasing the amount of trusted code.

5.3 Distributed web server

We have taken the HiStar web server described above, and used DStar to turn it into a three-tiered web application, as shown in Figure 8. HTTPS front-end servers run components responsible for accepting client connections and handling the HTTP protocol: the *launcher*, *SSLd*, *RSAd*, and *htpd*. Application servers run application code to execute requests. Finally, user data servers store private user data and perform user authentication.

HTTPS and application servers are largely stateless, making it easy to improve overall performance by adding more physical machines. This is an important consideration for complex web applications, where simple tasks such as generating a PDF document can easily consume 100 milliseconds of CPU time (using GNU ghostscript). User data servers can also be partitioned over multiple machines, by keeping a consistent mapping from each individual user to the particular user data server responsible for their data. Our prototype has a statically-configured mapping from users to user data servers, replicated on each HTTPS front-end server.

Our distributed web server has no central authority, and all data servers are mutually distrustful. For a web service that generates tax forms, this allows multiple companies to each provide their own data server. While each company may trust the web service to generate one employee's tax form, no company trusts anyone other than themselves with all of their employee data.

Similarly, different parts of a single user's data can be handled by different front-end and application servers. For example, if payment were required for accessing a tax document, separate front-end and application servers could be used to process credit card transactions. Even if those servers were to be compromised, user tax information would be handled by different front-end and application servers, and could remain secure. Conversely, if the servers handling tax data were compromised, credit card data would not necessarily be disclosed.

Interactions between components on a single machine are unchanged, with the same security properties as before. Interactions between components on different machines, on the other hand, maintain the same structure and security properties as before, but now go through the exporters on the respective machines, thereby making the exporters a part of the trusted code base. Communicating over DStar typically requires three things for each category involved: a mapping from the local category to a DStar category, certificates proving that the remote exporter is trusted by that DStar category, and a mapping from the DStar category to a local category on the remote machine, as we will describe next.

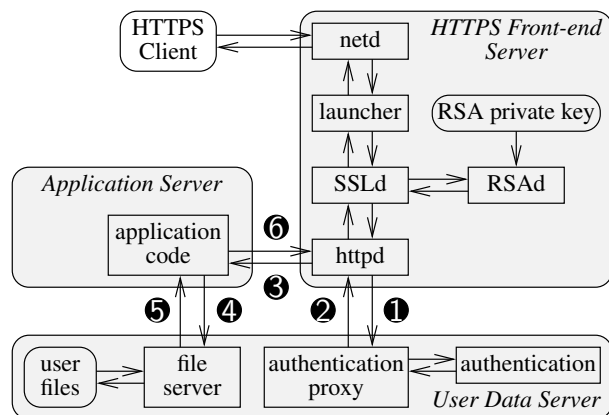


Figure 8: Structure of our web server running on multiple HiStar machines. Shaded boxes represent physical machines. Circled numbers indicate the order in which DStar messages are sent between machines. Not shown are DStar messages to create mappings on remote machines. Components have largely the same labels as in Figure 6.

The one exception where the distributed web server structurally differs from the one that runs on a single machine is authentication. We have not ported the three-phase HiStar authentication service to run over DStar yet; instead, we introduced a trusted authentication proxy to invoke the HiStar authentication service locally on the data server. *httpd* trusts the authentication proxy to keep the user's password secure, but guarded invocation ensures that the user's password is only passed to the correct authentication proxy process.

These changes to the HiStar web server added 740 lines of C++ code: 280 lines to *httpd*, a 140-line trusted authentication proxy, a 220-line untrusted RPC server to launch application code, and a 100-line file server.

To use DStar, applications, such as our distributed web server, must explicitly manage two important aspects of the distributed system. First, applications must explicitly define trust between the different machines in the distributed system, by creating and distributing the appropriate delegation certificates. Second, applications need to explicitly allocate resources, such as containers and category mappings, on different machines to be able to communicate between them and execute code remotely. The next two subsections describe how the distributed web server addresses these issues.

5.4 Trust management

The key trust relation in our distributed web server concerns the individual users' secrecy and integrity categories, or in other words, which machines are authorized to act on behalf of which users. All user categories in our design are initially created by, and therefore trust, the exporter on that user's data server. When the authentication proxy receives the correct user password (Step 1 in Figure 8), it asks the local exporter to create a short-lived delegation certificate, valid only for a few minutes, for

the user's secrecy and integrity categories to the exporter on *httpd*'s front-end machine. Short-lived certificates ensure that, even if some machines are compromised, they can only subvert the security of users that are currently using, or have recently used those machines. The authentication proxy sends these certificates to *httpd* in Step 2, and grants it ownership of the user's categories using the ownership field of the message.

Although *httpd* receives ownership of the user's categories, it does not use it directly. Instead, *httpd* passes ownership of the user's categories to the application server (Step 3), where application code uses it to communicate with the user data server (Step 4). *httpd* asks the exporter on its front-end machine to delegate its trust of these categories to the application server. To be considered valid, the delegation certificates created by the front end's exporter must be presented together with a chain of certificates up to the category's creator—the user data server—proving that the front-end machine was authorized to delegate trust in the first place. Since this chain includes the initial, short-lived certificate from the authentication proxy, malicious exporters cannot extend the amount of time they can act on the user's behalf, as long as the clock on the user data server does not go back.

5.5 Resource management

The distributed web server must also explicitly provide memory and CPU resources for all messages and processes. Since the *launcher* drives the execution of user requests, it requires such resources on all other machines. We use a special integrity category, r_i , to manage access to these resources, and each application and data server has an initial container labeled $\{r_i\}$, known to the *launcher*. The *launcher* owns r_i , giving it access to the initial container on those machines. We will describe later how this system is bootstrapped.

When the *launcher* starts *httpd*, it grants it ownership of r_i , and gives it the names of these initial containers on all other servers. When *httpd* talks to the authentication proxy in Step 1, for example, it uses the initial container on the corresponding user data server, along with its ownership of r_i , to send the request message.

Because HiStar labels all containers, the web server must take care to set the appropriate labels. Consider Step 4, when the application code wants to communicate with the file server. Although *httpd* could grant the application code ownership of r_i , the application code would not be able to use the initial container labeled $\{r_i\}$ on the data server, because the application code is labeled $\{u_s\}$ and cannot write to that container. Thus, *httpd* pre-allocates a sub-container with a label of $\{u_s, u_i\}$ on the user data server, using that machine's resource allocation service, and passes this container's name to the application code in Step 3. The application code can then use

this container to communicate with the file server, without being able to leak private user data through memory exhaustion.

Although in our prototype the application communicates with only one data server per user, a more complex application can make use of multiple data servers to handle a single request. Doing so would require the application code to have delegation certificates and access to containers on all of the data servers that it wants to contact. To do this, *httpd* could either pre-allocate all such delegations and containers ahead of time, or provide a callback interface to allocate containers and delegations on demand. In the latter case, *httpd* could rate-limit or batch requests to reduce covert channels.

5.6 Bootstrapping

When adding a new machine to our distributed web server, a bootstrapping mechanism is needed to gain access to the new machine's memory and CPU resources. For analogy, consider the process of adding a new machine to an existing Linux cluster. An administrator would install Linux, then from the console set a root password, configure an ssh server, and (if diligent about security) record the ssh host key to enter on other machines. From this point on, the administrator can access the new machine remotely, and copy over configuration files and application binaries. The ability to safely copy private data to the new machine stems from knowing its ssh host key, while the authority to access the machine in the first place stems from knowing its root password.

To add a new physical machine to a DStar cluster requires similar guarantees. Instead of an ssh host key, the administrator records the exporter's public key, but the function is the same, namely, for other machines to know they are talking to the new machine and not an impostor. However, DStar has no equivalent of the root password, and instead uses categories.

In fact, the root password serves two distinct purposes in Linux: it authorizes clients to allocate resources such as processes and memory on the new machine—i.e., to run programs—and it authorizes the programs that are run to access and modify data on the machine. DStar splits these privileges amongst different categories.

When first setting up the DStar cluster, the administrator creates an integrity category r_i^1 on the first HiStar machine (superscript indicates the machine), and a corresponding DStar category r_i that we mentioned earlier. r_i represents the ability to allocate and deallocate all memory and processes used by the distributed web server on any machine. It can be thought of as the “resource allocation root” for this particular application. However, there is no equivalent “data access root.” Instead, different categories protect different pieces of data.

In configuring a new machine, the administrator's goal is to gain access to the machine's resources over the network, using ownership of r_i . The new machine's exporter initially creates a local category r_i^n and a container labeled $\{r_i^n\}$ that will provide memory and CPU resources. To access this container, the administrator needs to establish a mapping between r_i and r_i^n on the new machine.

To do this, the administrator enters the name of r_i when starting the exporter on the new machine, which then creates a mapping between r_i and r_i^n . The exporter periodically broadcasts a signed message containing this mapping and the root container's ID, so the administrator need not manually transfer them.

With this mapping, processes that own r_i can now copy files to the new machine and execute code there, much as the ssh client can on Linux if it knows the new machine's root password. However, a process that also owns other categories can use them to create files that cannot be read or written by owners of r_i alone.

The current bootstrap procedure is tedious, requiring the manual transfer of category names and public keys. In the future, we envisage a setup utility that uses a password protocol like SRP [24] to achieve mutual authentication with an installation daemon, to automate the process. Alternatively, hardware attestation, such as TCPA, could be used to vouch that a given machine is running HiStar and a DStar exporter with a particular public key.

5.7 Replication

Having added a new machine to the DStar cluster, the administrator needs to securely replicate the web server onto it, and in particular, transfer the SSL private key to start a new *RSAd* process. We use a special replication daemon to do this, which ensures that the private key is only revealed to an *RSAd* binary on the remote machine.

To replicate *RSAd*, the administrator provides this daemon with a public key of the new machine, and access to a container on it (such as by granting it ownership of r_i). The replication daemon uses the mapping service to create a new category rsa_s^n on the new machine, which will protect the private key there. To ensure that the private key is not passed to the wrong process, the replication daemon uses guarded invocation to invoke an authentic *RSAd* process on the new machine with ownership of rsa_s^n , and passes it the private key protected with rsa_s^n . Note that the administrator, who granted the replication daemon access to a container on the new machine, cannot read the private key that is now stored there, because he does not own rsa_s^n .

Both the replication daemon and the guarded invocation service, consisting of 120 and 200 lines of C++ code respectively, must be trusted to keep the private key secure, in addition to *RSAd*. A similar mechanism is used to start the *launcher*. HiStar's system-wide persistence

eliminates the need for a trusted process to start *RSAd* and *launcher* when the machine reboots.

5.8 Heterogeneous systems

To illustrate how DStar facilitates incremental deployment, we show how Linux can use HiStar or Flume to execute untrusted perl code with strong security guarantees. We implemented a DStar RPC server on HiStar and on Flume that takes the source code of a perl script and input data, executes the script on that input, and returns perl's exit code and output. DStar translates information flow restrictions specified by the caller into HiStar or Flume labels, which are then enforced by the operating system.

This service can be used by an existing Linux web server to safely execute untrusted perl code. The Linux machine can specify how different instances of perl can share data, by specifying the policy using secrecy and integrity categories in the request's label. To ensure each request is processed in complete isolation, fresh secrecy and integrity categories can be used for each request. If different scripts running on behalf of the same user need to share data, such as by storing it in the file system on a HiStar machine, the same categories should be used for each request made on behalf of a given user.

DStar also allows building distributed applications using a mix of operating systems, such as both Flume and HiStar. This may be useful when no single operating system is a perfect fit in terms of security or functionality for every part of the application. However, this paper does not evaluate any such applications.

6 EVALUATION

To quantify the overheads imposed by DStar, we present a detailed breakdown of DStar's data structure sizes and measure the cost of generating and verifying certificate signatures. To put these costs in perspective, we evaluate the performance of our web server and perl service under various conditions. The overheads introduced by both HiStar and DStar in the web server are acceptable for compute-intensive web applications such as PDF generation, and the performance is close to that of a Linux system. On the other hand, our web server delivers static content much more slowly than Linux.

Benchmarks were run on 2.4GHz Intel Xeon machines with 4MB CPU caches, 1GB of main memory, and 100Mbps switched Ethernet. For web server comparison, we used Apache 2.2.3 on 64-bit Ubuntu 7.04 with kernel version 2.6.20-15-generic. This Linux web server forked off a separate application process to handle every client request. The PDF workload used a2ps version 4.13b and ghostscript version 8.54. Xen experiments used Xen version 3.0 and kernel 2.6.19.4-xen for all domains. The *netd* TCP/IP stack was running a HiStar user-mode port of Linux kernel 2.6.20.4. Web servers used

Data structure	Raw bytes	Compressed bytes
Public key	172	183
Category name	184	195
Category mapping	208	219
Unsigned delegation certificate	548	384
Signed delegation certificate	720	556
Unsigned address certificate	200	203
Signed address certificate	376	379
Null message (1)	200	194
Empty message (2)	1348	623

Figure 9: Size of DStar data structures. Delegation certificate delegates a category to another exporter, and is signed by the category's creator. Null message (1) has an empty label and no mappings, delegations, or payload. Empty message (2) has a label consisting of one category, and includes one delegation certificate, one mapping, and an empty payload. The compressed column shows the potential reduction in size that can be achieved by compressing the data structures using zlib.

Operation	Time (msec)
Sign a delegation certificate	1.37
Verify a delegation certificate	0.012
Sign an address certificate	1.35
Verify an address certificate	0.011
Null RPC on same machine	1.84

Figure 10: Microbenchmarks measuring the time to sign and verify certificates, and the round-trip time to execute a null RPC request on one machine, with an empty label and no delegations or mappings.

OpenSSL 0.9.8a with 1024-bit certificate keys; DStar exporters used 1280-bit Rabin keys.

6.1 Protocol analysis

Figure 9 shows the detailed sizes of DStar data structures as implemented in our prototype. The main source of space overhead in DStar messages is the public keys used to name categories and exporters. However, public keys are often repeated multiple times in the same message. For example, user secrecy and integrity categories are often created by the same exporter, and therefore share the same public key. Moreover, all of the delegations included in a message typically mention the same public key of the sending exporter. As a result, compressing messages results in significant space savings, as shown in the "compressed" column; however, our current prototype does not use compression. Storing only a hash of the public key in a category name can reduce its size, but would likely not reduce the size of a compressed message: delegation certificates in the same message are likely to include the entire public key (in order to verify certificate signatures), and with compression, there is little overhead for including a second copy of the same public key in the category name. Compressing the entire TCP session between two exporters, prior to encryption, is likely to generate further space savings, since the public keys of the two exporters are likely to be mentioned in each message. However, stream compression can lead to covert channels: the throughput observed by one process reveals the similarity between its messages and those sent by other processes.

System	PDF workload		cat workload	
	throughput	latency	throughput	latency
Linux Apache	7.6	137	110.3	15.7
HS PS+Auth	6.3	161	37.8	30.8
HS PS	6.6	154	49.5	24.6
HS no PS	6.7	150	71.5	19.1
DS on one machine	5.2	194	17.0	63.0
DS on multiple machines	varies	511	varies	345

Figure 11: Maximum throughput (requests/sec) and minimum latency (msec) for the PDF and cat workloads on one machine. “HS PS+Auth” ran the HiStar web server from Section 5.1. “HS PS” ran the same web server without HiStar’s user authentication service. “HS no PS” ran the same web server without privilege separation, with all components in a single address space. “DS” refers to the distributed web server.

The CPU overhead of public key operations on DStar certificates is shown in Figure 10. The Rabin cryptosystem used by DStar provides relatively fast signature verification but more expensive signature generation, which may be a reasonable trade-off for applications that do not change trust relations with each message. The overhead of issuing a local RPC, with an empty label and no mappings or delegations, is also shown in Figure 10.

6.2 Application performance

To evaluate the performance of applications running on our web server, we approximate a realistic service using a PDF generation workload. For each request, an HTTPS client connects to the server and supplies its user name and password. The web server generates a 2-page PDF document based on an 8KB user text file stored on the server, and sends it back to the client. Clients cycle through a small number of users; since the web server does not cache any authentication state, this does not skew performance. With multiple front-end servers, clients use a round-robin selection policy. We measure the minimum latency observed when a single client is active, and the maximum throughput achieved with the optimal number of clients; the optimal number of clients is generally proportional to the number of servers.

Figures 11 and 12 show the results of this experiment. A non-privilege-separated web server running on HiStar provides 12% less throughput than Linux; the difference is in part due to the overhead of fork and exec on HiStar. Privilege separation of the web server on HiStar imposes a 2% penalty, and HiStar’s user authentication service, which requires eight gate calls, reduces throughput by another 5%. Running the distributed web server on a single machine shows the overhead imposed by DStar, although such a setup would not be used in practice. The throughput of the PDF workload scales well with the total number of servers.

6.3 Web server overhead

To better examine the overhead of our web server, we replaced the CPU-intensive PDF workload with cat, which just outputs the same 8KB user file; the results are also shown in Figures 11 and 12. Apache has much

Calling machine	Linux	HiStar	Linux	Linux	Linux
Execution machine	same	same	Linux	HiStar	Flume
Communication	none	none	TCP	DStar	DStar
Throughput, req/sec	505	334	160	67	61
Latency, msec	2.0	3.0	6.3	15.7	20.6

Figure 13: Throughput and latency of executing a “Hello world” perl script in different configurations.

higher throughput than the HiStar web server both with and without privilege separation. Though Apache serves static content better without using cat, we wanted to measure the overhead of executing application code. Our web server’s lower performance reflects that its design is geared towards isolation of complex application code; running simple applications incurs prohibitively high overhead. Nonetheless, the distributed web server still scales with the number of physical machines.

6.4 Privilege separation on Linux

Is it possible to construct a simpler, faster privilege-separated web server on Linux that offers similar security properties? We constructed a prototype, running separate *launcher*, *SSLd*, *RSAd*, and Apache processes, using *chroot* and *setuid* to isolate different components from each other. As in the earlier Apache evaluation, a fresh application process was forked off to handle each client request. This configuration performed similar to a monolithic Apache server. However, to isolate different user’s application code from one another, Apache (a 300,000 line program) needs access to *setuid*, and needs to run as root, a step back in security. We can fix this by running Apache and application code in a Xen VM; this reduces the throughput of the PDF workload to 4.7 req/sec. Even this configuration cannot guarantee that malicious application code cannot disclose user data; doing so would require one VM per request, a fairly expensive proposition. This suggests that the complexity and overhead of HiStar’s web server may be reasonable for the security it provides, especially in a distributed setting.

6.5 Heterogeneous systems

Figure 13 shows the latency and throughput of running a simple “untrusted” perl script that prints “Hello world” on Linux, on HiStar, on Linux invoked remotely using a simple TCP server, and on HiStar and Flume invoked remotely using DStar from Linux. A fresh secrecy category is used for each request in the last two cases. This simple perl script provides a worst-case scenario, incurring all of the overhead of perl for little computation; more complex scripts would fare much better. The lower perl performance on HiStar is due to the overhead of emulating fork and exec. DStar incurs a number of round-trips to allocate a secrecy category and create a container for ephemeral call state, which contributes to a significantly higher latency. Comparing the throughput on HiStar run-

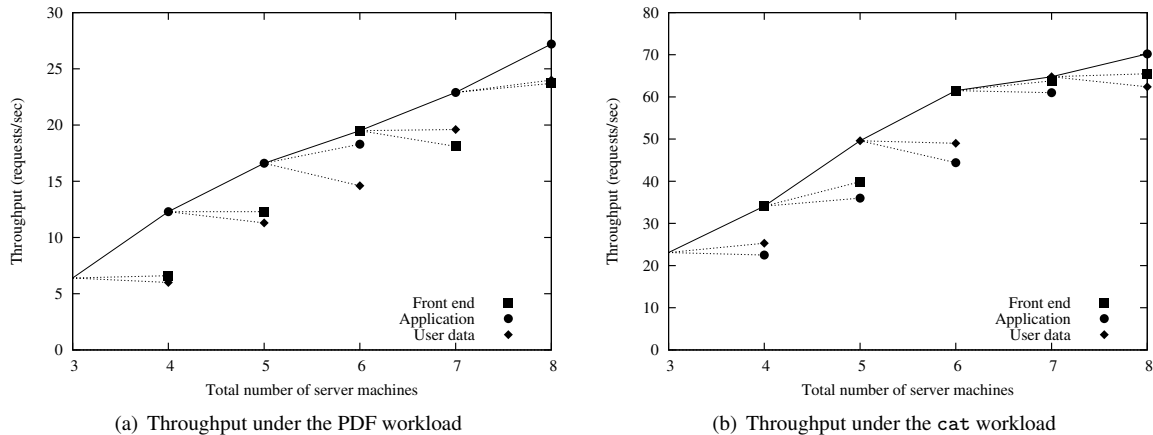


Figure 12: Maximum throughput achieved by the DStar web server running on multiple machines. The initial point on the left represents three machines: one front-end server, one application server, and one user data server. Subsequent points reflect the total throughput with an extra front-end server, application server, or user data server added, as indicated by the point symbol. The process is repeated for the highest-performing configuration at each step. The best use of an additional server machine is indicated by a solid line, and others by a dashed line.

ning locally and remotely shows that DStar adds an overhead of 12 msec of CPU time per request, which may be an acceptable price for executing arbitrary perl code from a Linux machine with well-defined security properties.

7 RELATED WORK

Flume [12] controls information flow in a fully-trusted centralized cluster sharing an NFS server and tag registry. Flume does not allow applications to directly communicate between machines, define their own trust relations, or share data across administrative domains. On the other hand, applications in DStar have complete control over trust relations for their data, and can communicate between any machines that speak the DStar protocol. Flume's centralized design limits scalability to small, fully-trusted local clusters, and cannot withstand any machine compromises. DStar could be used to connect multiple Flume clusters together without any inherent centralized trust or scalability bottlenecks.

Capability-based operating systems, such as KeyKOS [4] and EROS [17], can provide strict program isolation on a single machine. A DStar exporter could control information flow on a capability-based operating system by ensuring that processes with different labels had no shared capabilities other than the exporter itself, and therefore could not communicate without the exporter's consent.

Shamon [14] is a distributed mandatory access control system that controls information flow between virtual machines using a shared reference monitor. DStar avoids any centralized reference monitor for security and scalability. Shamon tracks information flow at the granularity of x86 virtual machines, making it impractical to track each user's data. DStar running on HiStar can apply policies to fine-grained objects such as files or threads.

A number of systems, including Taos [23] and Amoeba [19], enforce discretionary access control in a distributed system, often using certificates [3]. None of them can control information flow, as a malicious program can always synthesize capabilities or certificates to contact a colluding server. The Taos *speaks-for* relation inspired the much simpler DStar *trusts* relation, used to define discretionary privileges for different categories between exporters.

Multi-level secure networks [2, 9, 15, 18] enforce information flow control in a trusted network, but provide very coarse-grained trust partitioning. By comparison, DStar functions even in an untrusted network such as the Internet, at the cost of introducing some inherent covert channels, and allows fine-grained trust to be explicitly configured between hosts. Using a secure, trusted network would reduce covert channels introduced by DStar.

Unlike multi-level secure networks, DStar does not allow labeling a machine without giving it ownership privileges. Providing a non-empty machine label would require a trusted component to act as a proxy for the machine, ensuring that any packets sent or received by the machine are consistent with its current label. This can be done either with support from the network, or by explicitly forwarding messages through a proxy trusted to maintain the labels of machines it is proxying.

Secure program partitioning [25] partitions a single program into sub-programs that run on a set of machines specified at compile time with varying trust, to uphold an overall information flow policy. DStar is complementary, providing mechanisms to enforce an overall information flow policy without restricting program structure, language, or partitioning mechanism. DStar could execute secure program partitioning's sub-programs in a distributed system without trusting the partitioning com-

piler. Secure program partitioning has a much larger TCB, and relies on trusted external inputs to avoid a number of difficult issues addressed by DStar, such as determining when it is safe to connect to a given host at runtime, when it is safe to allocate resources like memory, and bootstrapping.

Jif [16] provides decentralized information flow control in a Java-like language. Although its label model differs from DStar's, a subset of Jif labels can be expressed by DStar. DStar could provide more fine-grained information flow tracking by enforcing it with a programming language like Jif rather than with an operating system.

Jaeger et al [11] and KDLM [7] associate encryption keys with SELinux and Jif labels, respectively, and exchange local security mechanisms for encryption, much like DStar. These approaches assume the presence of an external mechanism to bootstrap the system, establish trust, and define mappings between keys and labels—difficult problems that are addressed by DStar. Moreover, these approaches configure relatively static policies and trust relations at compile time; DStar allows any application to define new policies and trust at runtime. DStar transfers the entire label in each message, instead of associating labels with keys, since applications never handle ciphertexts directly. An application that receives a ciphertext but not the corresponding key may still infer confidential information from the size of the ciphertext.

8 SUMMARY

DStar is a framework for securing distributed systems by specifying end-to-end information flow constraints. DStar leverages the security label mechanisms of DIFC-based operating systems. Each machine runs an exporter daemon, which translates between local OS labels and globally-meaningful DStar labels. Exporters ensure it is safe to communicate with other machines, but applications define the trust relations. Self-certifying categories avoid the need for any fully-trusted machines. Using DStar, we built a highly privilege-separated, three-tiered web server in which no components are fully-trusted and most components cannot compromise private user data even by acting maliciously. The web server scales well with the number of physical machines.

ACKNOWLEDGMENTS

We thank Michael Walfish, Siddhartha Annapureddy, Pavel Brod, Antonio Nicolosi, Junfeng Yang, Alex Yip, the anonymous reviewers, and our shepherd, Ken Birman, for their feedback. This work was funded by NSF Cybertrust award CNS-0716806, by joint NSF Cybertrust and DARPA grant CNS-0430425, by the DARPA Application Communities (AC) program as part of the VERNIER project at Stanford and SRI International, and by a gift from Lightspeed Venture Partners.

REFERENCES

- [1] O. Aciğmez, Çetin Kaya Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. *Cryptology ePrint Archive*, Report 2006/351, 2006. <http://eprint.iacr.org/>.
- [2] J. P. Anderson. A unification of computer and network security concepts. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 77–87, Oakland, CA, 1985.
- [3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996.
- [4] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.
- [5] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proc. of the 12th USENIX Security Symposium*, August 2003.
- [6] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. of the 15th USENIX Security Symposium*, Vancouver, BC, 2006.
- [7] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *16th IEEE Computer Security Foundations Workshop*, pages 170–186. IEEE Computer Society, 2003.
- [8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [9] D. Estrin. Non-discretionary controls for inter-organization networks. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 56–61, Oakland, CA, 1985.
- [10] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proc. of the 2002 Ottawa Linux Symposium*, pages 479–495, June 2002.
- [11] T. Jaeger, K. Butler, D. H. King, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for mandatory access control across systems. In *Proc. of the 2nd International Conference on Security and Privacy in Communication Networks*, August 2006.
- [12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st SOSP*, Stevenson, WA, October 2007.
- [13] D. Mazières. A toolkit for user-level file systems. In *Proc. of the 2001 USENIX*, pages 261–274, June 2001.
- [14] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamoon: A system for distributed mandatory access control. In *Proc. of the 22nd Annual Computer Security Applications Conference*, Washington, DC, USA, 2006. IEEE Computer Society.
- [15] J. McHugh and A. P. Moore. A security policy and formal top-level specification for a multi-level secure local area network. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 34–39, Oakland, CA, 1986.
- [16] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOCS*, 9(4):410–442, October 2000.
- [17] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th SOSP*, December 1999.
- [18] D. P. Sidhu and M. Gasser. A multilevel secure local area network. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 137–143, Oakland, CA, 1982.
- [19] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mulender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, December 1990.
- [20] Think Computer Corporation. Identity crisis. <http://www.thinkcomputer.com/corporate/whitepapers/identitycrisis.pdf>.
- [21] S. VanDeBogart, P. Efstathiopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM TOCS*, 25(4):11:1–43, December 2007.
- [22] C. Weissman. Security controls in the ADEPT-50 time-sharing system. In *Proc. of the 35th AFIPS Conference*, pages 119–133, 1969.
- [23] E. P. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM TOCS*, 12(1):3–32, 1994.
- [24] T. Wu. The secure remote password protocol. In *Proc. of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.
- [25] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. of the 18th SOSP*, pages 1–14, October 2001.
- [26] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, November 2006.

Wedge: Splitting Applications into Reduced-Privilege Compartments

Andrea Bittau Petr Marchenko Mark Handley Brad Karp
University College London

Abstract

Software vulnerabilities and bugs persist, and so exploits continue to cause significant damage, particularly by divulging users' sensitive data to miscreants. Yet the vast majority of networked applications remain monolithically structured, in stark contravention of the ideal of least-privilege partitioning. Like others before us, we believe this state of affairs continues because today's operating systems offer isolation primitives that are cumbersome. We present *Wedge*, a system well suited to the splitting of complex, legacy, monolithic applications into fine-grained, least-privilege compartments. *Wedge* consists of two synergistic parts: OS primitives that create compartments with *default-deny* semantics, which force the programmer to make compartments' privileges explicit; and *Crowbar*, a pair of run-time analysis tools that assist the programmer in determining which code needs which privileges for which memory objects. By implementing the *Wedge* system atop Linux, and applying it to the SSL-enabled Apache web server and the OpenSSH login server, we demonstrate that *Wedge* allows fine-grained compartmentalization of applications to prevent the leakage of sensitive data, at acceptable performance cost. We further show that *Wedge* is powerful enough to prevent a subtle man-in-the-middle attack that succeeds on a more coarsely privilege-separated Apache web server.

1 Introduction

In the era of ubiquitous network connectivity, the consequences of vulnerabilities in server software grow ever more serious. The *principle of least privilege* [16] entails dividing the code into compartments, each of which executes with the minimum privileges needed to complete its task. Such an approach not only limits the harm malicious injected code may cause, but can also prevent bugs from accidentally leaking sensitive information.

A programmer frequently has a good idea which data manipulated by his code is sensitive, and a similarly good idea which code is most risky (typically because it handles user input). So why do so few programmers of networked software divide their code into minimally privileged compartments? As others have noted [2, 6], one reason is that the isolation primitives provided by today's operating systems grant privileges by default, and so are cumbersome to use to limit privilege.

Consider the use of processes as compartments, and the behavior of the *fork* system call: by default a child process inherits a clone of its parent's memory, including any sensitive information therein. To prevent such implicit granting of privilege to a child process, the parent can scrub all sensitive data from memory explicitly. But doing so is brittle; if the programmer neglects to scrub even a single piece of sensitive data in the parent, the child gains undesired read privileges. Moreover, the programmer may not even know of all sensitive data in a process's memory; library calls may leave behind sensitive intermediate results.

An obvious alternative is a *default-deny* model, in which compartments share no data unless the programmer explicitly directs so. This model avoids unintended privilege sharing, but the difficulties lie in how precisely the programmer can request data sharing, and how he can identify which data must be shared. To see why, consider as an example the user session-handling code in the Apache web server. This code makes use of over 600 distinct memory objects, scattered throughout the heap and globals. Just identifying these is a burden. Moreover, the usual primitives of *fork* to create a compartment, *exec* to scrub all memory, and inter-process communication to share only the intended 600 memory objects are unwieldy at best in such a situation.

In this paper, we present *Wedge*, a system that provides programming primitives to allow the creation of compartments with default-deny semantics, and thus avoids the risks associated with granting privileges implicitly upon process creation. To abbreviate the explicit granting of privileges to compartments, *Wedge* offers a simple and flexible memory tagging scheme, so that the programmer may allocate distinct but related memory objects with the same tag, and grant a compartment memory privileges at a memory-tag granularity.

Because a compartment within a complex, legacy, monolithic application may require privileges for many memory objects, *Wedge* importantly includes *Crowbar*, a pair of tools that analyzes the run-time memory access behavior of an application, and summarizes for the programmer which code requires which memory access privileges.

Neither the primitives nor the tools alone are sufficient. Default-deny compartments demand tools that make it feasible for the programmer to identify the mem-

ory objects used by a piece of code, so that he can explicitly enumerate the correct privileges for that code's compartment. And run-time analysis reveals memory privileges that a programmer should consider granting, but cannot enumerate those that should be denied; it thus fits best with default-deny compartments. The synergy between the primitives and tools is what we believe will yield a system that provides fine-grained isolation, and yet is readily usable by programmers.

To demonstrate that Wedge allows fine-grained separation of privileges in complex, legacy, monolithic applications, we apply the system to the SSL-enabled Apache web server and the OpenSSH remote login server for Linux. Using these two applications, we demonstrate that Wedge can protect against several relatively simple attacks, including disclosure of an SSL web server's or OpenSSH login server's private key by an exploit, while still offering acceptable application performance. We further show how the fine-grained privileges Wedge supports can protect against a more subtle attack that combines man-in-the middle interposition and an exploit of the SSL web server.

2 A Motivating Example

To make least-privilege partitioning a bit more concrete, consider how one might partition a POP3 server, as depicted in Figure 1. One can split the server into three logical compartments: a client handler compartment that deals with user input and parses POP3 commands; a login compartment that authenticates the user; and an e-mail retriever compartment that obtains the relevant e-mails. The login compartment will need access to the password database, and the e-mail retrieval compartment will need access to the actual e-mails; these two are *privileged* in that they must run with permissions that allow them to read these data items. The client handler, however, is a target for exploits because it processes untrusted network input. It runs with none of these permissions, and must authenticate users and retrieve e-mails through the restricted interface to the two privileged compartments.

Because of this partitioning, an exploit within the client handler cannot reveal any passwords or e-mails, since it has no access to them. Authentication cannot be skipped since the e-mail retriever will only read e-mails of the user id specified in *uid*, and this can only be set by the login component. One must, however, ensure that code running in the privileged compartments cannot be exploited. This task is simplified since the code that requires audit has been greatly reduced in size by factoring out the client handler, which most likely consists of the bulk of the code. A typical monolithic implementation would combine the code from all three compartments into a single process. An exploit anywhere in the code

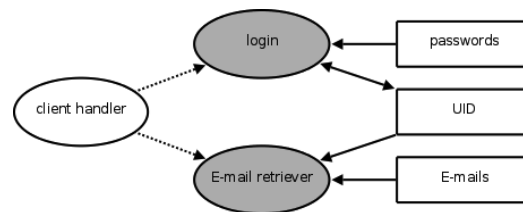


Figure 1: A partitioned POP3 server. Ovals represent code segments and those shaded are privileged. Dashed arrows between ovals indicate the capability of invoking a privileged code segment. Boxes represent memory regions and a one-way arrow from memory to code indicates read permission; two-way arrows indicate read-write.

could cause anything in the process's memory, including passwords and e-mails, to be leaked. Hence, partitioning can reduce the impact of exploits.

Our aim in building Wedge is to allow the programmer to create an arbitrary number of compartments, each of which is granted no privileges by default, but can be granted fine-grained privileges by the programmer. To the extent possible, we would like the primitives we introduce to resemble familiar ones in today's operating systems, so that programmers find them intuitive, and minimally disruptive to introduce into legacy application code.

Wedge achieves this goal with three isolation primitives, which naturally apply to the POP3 example:

Sthreads An sthread defines a compartment within an application. The programmer assigns access rights to memory and other resources per-sthread. In Figure 1, the unshaded oval is an sthread.

Tagged Memory When allocating memory, the programmer may mark that memory with a single tag. Access rights to memory are granted to sthreads in terms of these tags (e.g., "read/write for memory with tag *t*"). In Figure 1, each rectangle is a memory region with a distinct tag.

Callgates A callgate executes code with different privileges than its caller. An sthread typically runs with the least privilege possible. When it must perform an operation requiring enhanced privilege, it invokes a callgate that performs the operation on its behalf. A callgate defines a narrow interface to privileged code and any sensitive data it manipulates, and thus allows improved adherence to least-privilege partitioning. In Figure 1, each shaded oval is a callgate.

3 System Design

We now describe the operating system primitives that Wedge introduces, and thereafter, the Crowbar run-time analysis tools that ease programmer use of these primitives.

3.1 Sthreads

An sthread is the embodiment of a compartment in a partitioned application. It consists of a thread of control and an associated security policy that specifies:

- The memory tags the sthread may access, and the permissions for each (read, read-write, copy-on-write).
- The file descriptors the sthread may access, and the permissions for each (read, write, read-write).
- The callgates the sthread may invoke.
- A UNIX user id, root directory, and an SELinux policy [10], which limits the system calls that may be invoked.

A newly created sthread holds no access rights by default, apart from copy-on-write access to a pristine snapshot of the original parent process's memory, taken just *before* the execution of *main*, including the sthread's (as yet unused) private stack and heap. (We explain further in Section 4.1 why this snapshot does not contain sensitive data, and is essential for correct execution.) It is unable to access any other memory nor any file descriptor from its creator, nor invoke any callgates or system calls. Note that all system calls retain the standard in-kernel privilege checks, based on the caller's user id, root directory, and SELinux policy. A parent may grant a child sthread access to its resources simply by attaching an appropriate security policy to the child sthread when creating it.

An sthread can only create a child sthread with equal or lesser privileges than its own. Specifically, a parent can only grant a child access to subsets of its memory tags, file descriptors, and authorized callgates. Similarly, the *userid* and *filesystem root* of a child sthread can be changed only according to UNIX semantics (*i.e.*, only if the parent runs as superuser), and any changes in the SELinux policy must be explicitly allowed as domain transitions in the system-wide SELinux policy.

Because most CPUs do not support write-only memory permissions, Wedge does not allow them; the programmer must instead grant read-write permissions.

3.2 Tagged Memory

The programmer expresses memory privileges for stthreads in terms of *tags*, which are assigned to memory regions at allocation time. When he wishes to share tagged memory, he grants privileges for that tag to a newly created sthread. The tag namespace is flat, so privileges for one tag never imply privileges for other tags.

Programmers allocate tagged memory in two steps. First, the programmer must create a tag. This operation essentially allocates a memory segment and stores a mapping from the tag to the segment. Next, the programmer invokes a tagged memory allocation (*smalloc*):

he specifies a tag and a desired size, and thus allocates a buffer of that size from the segment with that tag. The ability to identify many memory regions with a single tag simplifies policy specification. Note that any memory an sthread allocates without using this two-stage process (tag, then *smalloc*) has no tag, and thus cannot be accessed by other stthreads: it cannot even be named in a security policy. In this way, computations performed on an sthread's stack or heap are by default strongly isolated from other stthreads. We use the standard hardware page protection mechanism to enforce the access permissions for tagged memory specified in an sthread's security policy.

When writing new applications, the mechanisms described so far suffice for tagging memory. But when partitioning existing applications, one may need to tag global variables, or convert many *malloc* calls within a function to use *smalloc* instead, which may not even be possible for allocations in binary-only libraries. We therefore provide two additional mechanisms for tagging memory, specifically tailored for splitting legacy applications. The first allows declaring globals with a tag, and works by placing all globals with the same tag in a distinct, contiguous section of the ELF binary. The second allows the programmer to specify that *all* calls to the standard C *malloc* between two points in a program should be replaced with *smalloc* calls with a particular tag. To do so, the programmer simply places calls to utility functions *smalloc_on* and *smalloc_off* at the desired start and end points, respectively. These aids greatly simplify the introduction of compartments; much of the work of partitioning consists of identifying and tagging memory correctly.

3.3 Callgates

A callgate is a portion of code that runs with different (typically higher) privileges than its caller. In our POP3 example in Figure 1, the login and e-mail retriever entities are callgates. Instantiating a callgate with access to sensitive data isolates the sensitive data from access by untrusted code, such as the sthread that parses network input in the POP3 server.

A callgate is defined by an entry point, a set of permissions, and a trusted argument supplied by the callgate's creator (usually a pointer into trusted memory), that the kernel will pass to the entry point when the callgate is invoked. The trusted argument allows the callgate's creator to pass the callgate input that cannot be tampered with by its caller. A callgate also inherits the *filesystem root* and *user id* of its creator. A callgate's permissions must be a subset of those of the sthread that creates the callgate. After a privileged sthread creates a callgate, it may spawn a child sthread with reduced privilege, but grant that child permission to invoke the callgate. Upon callgate invoca-

tion, a new sthread with the callgate's permissions is created, and begins execution at the entry point specified by the callgate's original creator. The caller blocks until the callgate terminates, and then collects any return values.

The dominant costs of invoking a short-lived callgate are those incurred creating and destroying the underlying sthread. For throughput-critical applications, we provide long-lived *recycled callgates*, which amortize their creation cost over many invocations. Because they are reused, recycled callgates do trade some isolation for performance, and must be used carefully; should a recycled callgate be exploited, and called by stthreads acting on behalf of different principals, sensitive arguments from one caller may become visible to another.

3.4 Crowbar: Partitioning Assistance

To identify the many memory dependencies between different blocks of code, and hence make default-deny partitioning primitives usable in practice, we provide *Crowbar*, a pair of Linux tools that assist the programmer in applying the primitives in applications. Broadly speaking, these tools analyze the run-time behavior of a legacy monolithic application to identify exactly which items in memory are used by which specific pieces of code, with what modes of access, and where all those items were allocated. This analysis suggests a set of privileges that appear required by a particular piece of code.

The programmer uses Crowbar in two phases. In Crowbar's run-time instrumentation phase, the *cb-log* tool logs memory allocations and accesses made by the target application. In Crowbar's analysis phase, the programmer uses the *cb-analyze* tool to query the log for complex memory access patterns during the application's run that are relevant when partitioning the application in accordance with least privilege.

First, *cb-log* produces a trace of all memory accesses. *cb-log* stores a complete backtrace for every memory read or write during execution, so that the programmer can determine the context of each access. These backtraces include function names and source filenames and line numbers. *cb-log* identifies global memory accesses by variable name and source code location; stack memory accesses by the name of the function in whose stack frame the access falls; and heap memory accesses by a full backtrace for the original *malloc* where the accessed memory was first allocated. This information helps programmers identify which globals to tag using our mechanisms, which stack allocations to convert to heap ones, and which specific *malloc* calls (revealed in our trace) to convert to *smalloc* calls.

Second, after *cb-log* produces a trace, the programmer uses *cb-analyze* to query it for specific, summarized information. The supported queries are:

- Given a procedure, what memory items do it *and all its descendants in the execution call graph* access during their execution, and with what modes of access? When the programmer wishes to execute a procedure in a least-privilege sthread, he uses this query to learn the memory items to which he must grant that sthread access, and with what permissions.
- Given a list of data items, which procedures use any of them? When the programmer wishes to create a callgate with elevated privileges to access sensitive data, he uses this query to learn which procedures should execute within the callgate.
- Given a procedure known to generate sensitive data, where do it and *all its descendants in the execution call graph* write data? When the programmer wishes to learn which data may warrant protection with callgates, he uses this query to identify the memory that should be kept private to that callgate. This query is particularly useful in cases where a single procedure (and its children) may generate large volumes of sensitive data; it produces data items of interest for queries of the previous type.

We stress that for the first and third query types, including children in the execution call graph makes *cb-analyze* particularly powerful; one often need not understand the complex call graph beneath a procedure to partition an application.

Crowbar is useful even after the initial partitioning effort. For example, after code refactoring, an sthread may stop functioning, as it may access additional memory regions not initially specified in its policy. We provide an *sthread emulation* library, which grants stthreads access to *all* memory, so that protection violations do not terminate stthreads. The programmer may use this library with Crowbar to learn of all protection violations that occur during a complete program execution.

Because Crowbar is trace-driven, the programmer will only obtain the memory permissions used during one particular run. To ensure coverage of as broad a portion of the application as possible, the programmer may generate traces by running the application on diverse innocuous workloads with *cb-log*, and running *cb-analyze* on the aggregation of these traces.

4 Implementation

Wedge's implementation consists of two parts: the OS isolation primitives, for Linux kernel version 2.6.19, and the userland Crowbar tools, *cb-log* and *cb-analyze*.

4.1 Isolation Primitives

Table 1 shows Wedge's programming interface.

Sthread-related calls	
int sthread_create(sthread_t *thrd, sc_t *sc, cb_t cb, void *arg); int sthread_join(sthread_t thrd, void **ret);	
Memory-related calls	
tag_t tag_new();	int tag_delete(tag_t);
void* smalloc(int sz, tag_t tag);	void sfree(void *x);
void smalloc_on(tag_t tag);	void smalloc_off();
BOUNDARY_VAR(def, id);	BOUNDARY_TAG(id);
Policy-related calls	
void sc_mem_add(sc_t *sc, tag_t t, unsigned long prot); void sc_fd_add(sc_t *sc, int fd, unsigned long prot); void sc_sel_context(sc_t *sc, char *sid);	
Callgate-related calls	
void sc_cgate_add(sc_t *sc, cg_t cgate, sc_t *cgsc, void *arg); void* cgate(cg_t cb, sc_t *perms, void *arg);	

Table 1: The Wedge programming interface.

Sthread-related Calls The programming interface for sthreads closely resembles that for pthreads, apart from the introduction of a security policy argument (*sc*). We implement sthreads as a variant of Linux processes. Rather than inheriting the entire memory map and all file descriptors from its parent, a newly spawned sthread inherits only those memory regions and file descriptors specified in the security policy. As with *fork*, the new sthread will have its own private signal handlers and file descriptor copies, so receiving a signal, closing a file descriptor, and exiting do not affect the parent.

Sthreads also receive access to a private stack and a private copy of global data. The latter represents the memory map of the application’s first-executed process, just before the calling of the C entry point *main*. This memory is vital to sthread execution, as it contains initialized state for shared libraries and the dynamic loader. It does not typically, however, contain any sensitive data, since the application’s code has yet to execute. In cases where statically initialized global variables are sensitive, we provide a mechanism (*BOUNDARY_VAR*) for tagging these, so that sthreads do not obtain access to them by default. Our implementation stores a copy of the pages of the program just before *main* is called, and marks these pages copy-on-write upon *main*’s invocation or any sthread creation.

Memory-related Calls *smalloc* and *sfree* mimic the usual *malloc* and *free*, except that *smalloc* requires specification of the tag with which the memory should be allocated. Tags are created using *tag_new*, a system call that behaves like anonymous *mmap*. Unlike *mmap*, *tag_new* does not merge neighboring mappings, as they may be used in different security contexts. Apart from creating a new memory area, *tag_new* also initializes internal bookkeeping structures used by *smalloc* and *sfree* on memory with that tag. The *smalloc* implementation is derived from *dlmalloc* [9].

Much of *tag_new*’s overhead comes from system call overhead and initializing the *smalloc* bookkeeping struc-

tures for that tag. We mitigate system call overhead by caching a free-list of previously deleted tags (*i.e.*, memory regions) in userland, and reusing them if possible, hence avoiding the system call. To provide secrecy, we scrub a tag’s memory contents upon tag reuse. Rather than scrubbing with (say) zeros, we copy cached, pre-initialized *smalloc* bookkeeping structures into it, and thus avoid the overhead of recomputing these contents.

As described in Section 3.2, *smalloc_on* and *smalloc_off* ease the tagging of heap memory. They convert any standard *malloc* which occurs between them into an *smalloc* with the *tag* indicated in *smalloc_on*. To implement this feature, Wedge intercepts calls to *malloc* and *free* using LD_PRELOAD, and checks the state of a global flag indicating whether *smalloc_on* is active; if so, *smalloc* is invoked, and if not, *malloc* is. In our current implementation, this flag is a single per-sthread variable. Thus, *smalloc_on* will not work if invoked recursively, and is neither signal- nor thread-safe. In practice, however, these constraints are not limiting. The programmer can easily save and restore the *smalloc_on* state at the start and end of a signal handler. Should a programmer need to use *smalloc_on* in recursive or thread-concurrent code (within the same sthread), he can easily save-and-restore or lock the *smalloc_on* state, respectively.

The *BOUNDARY_VAR* macro supports tagging of globals, by allowing each global declaration to include an integer *ID*, and placing all globals declared with the same *ID* in the same, separate, page-aligned section in the ELF binary. This allows for specific pages, in this case global variables, to be carved out of the data segment, if necessary. At runtime, the *BOUNDARY_TAG* macro allocates and returns a unique tag for each such *ID*, which the programmer can use to grant sthreads access to globals instantiated with *BOUNDARY_VAR*. This mechanism can be used to protect sensitive data that is statically initialized, or simply to share global data structures between sthreads.

Policy-related Calls These calls manipulate an *sc_t* structure, which contains an sthread policy; they are used to specify permissions for accessing memory and file descriptors. To attach an SELinux policy to an sthread, one specifies the SID in the form of *user:role:type* with *sc_sel_context*.

Callgate-related Calls *sc_cgate_add* adds permission to invoke a callgate at entry point *cgate* with permissions *cgsc* and trusted argument *arg* to a security policy. The callgate entry point, permissions and trusted argument are stored in the kernel, so that the user may not tamper with them, and are retrieved upon callgate invocation. When a parent adds permission to invoke a callgate to a security policy, that callgate is implicitly instantiated

when the parent binds that security policy to a newly created sthread.

To invoke a callgate, an sthread uses the *cgate* call, giving additional permissions *perms* and an argument *arg*. This argument will normally be created using *smalloc*, and the additional permissions are necessary so that the callgate may read it, as by default it cannot. Upon callgate invocation, the kernel checks that the specified entry point is valid, and that the sthread has permission to invoke the callgate. The permissions and trusted argument are retrieved from the kernel, and the kernel validates that the argument-accessing permissions are a subset of the sthread's current permissions.

Callgates are implemented as separate sthreads so that the caller cannot tamper with the callee (and vice-versa). Upon invocation, the calling sthread will block until the callgate's termination. Because the callgate runs using a different memory map (as a different sthread), the caller cannot exploit it by, e.g., invalidating dynamic loader relocations to point to shellcode, or by other similar attacks. Any signals delivered during the callgate's execution will be handled by the callgate. A caller may only influence a callgate through the callgate's untrusted argument.

We currently implement recycled callgates directly as long-lived sthreads. To invoke a recycled callgate, one copies arguments to memory shared between the caller and underlying sthread, wakes the sthread through a futex [3], and waits on a futex for the sthread to indicate completion.

4.2 Crowbar

Crowbar consists of two parts: *cb-log* traces memory access behavior at run-time, and *cb-analyze* queries the trace for summarized data. We only describe *cb-log*, as *cb-analyze* is essentially a text-search tool, and is thus straightforward. *cb-log* uses Pin [11]'s run-time instrumentation functionality. *cb-log* has two main tasks: tracking the current backtrace, and determining the original allocation site for each memory access.

To compute the backtrace, we instrument every function entry and exit point, and walk the saved frame pointers and return addresses on the stack, much as any debugger does. We thus rely on the code's compilation with frame pointers.

To identify original memory allocations, we instrument memory loads and stores. We also keep a list of *segments* (base and limit), and determine whether a memory access lies within a certain segment, and report the segment if so. There are three types of segments: globals, heap, and stack. For globals, we use debugging symbols to obtain the base and limit of each variable. For the heap, we instrument every *malloc* and *free*, and create a segment for each allocated buffer. For the stack, we use a

function's stack frame as the segment. Upon access, together with the segment name, we also log the offset being accessed within the segment. This offset allows the programmer to calculate and determine the member of a global or heap structure being accessed, or the variable within a stack frame being touched.

Our implementation handles *fork* and *pthreads* correctly; it clones the memory map and keeps separate backtraces for the former, and keeps a single memory map but different backtraces for the latter. *cb-log* also supports the sthread emulation library, by logging any memory accesses by an sthread for which insufficient permissions would normally have caused a protection violation. Because the sthread emulation library works by replacing sthreads by standard pthreads, our current implementation does not yet support copy-on-write memory permissions for emulated sthreads.

5 Applications

To validate the utility of Wedge, we apply it to introduce fine-grained, reduced-privilege compartments into two applications: the Apache/OpenSSL web server, and the OpenSSH remote login server. Because SELinux already provides a mechanism to limit system call privileges for sthreads, we focus instead on memory privileges in this paper. Thus, when we partition these two applications, we specify SELinux policies for all sthreads that explicitly grant access to *all* system calls.

5.1 Apache/OpenSSL

Our end-to-end goal in introducing compartments into Apache/OpenSSL is to preserve the confidentiality and integrity of SSL connections—that is, to prevent one user from obtaining the cleartext sent over another user's SSL connection, or from injecting content into another user's SSL connection.

We consider two threat models. In the first, simpler one, the attacker can eavesdrop on entire SSL connections, and can exploit any unprivileged compartment in the Apache/OpenSSL server. In the second, subtler one, the attacker can additionally interpose himself as a man-in-the-middle between an innocent client and the server. Let us consider each in turn. In what follows, we only discuss the SSL handshake as performed with the RSA cipher, but we expect the defenses we describe apply equally well to SSL handshakes with other ciphers.

5.1.1 Simple model (no interposition)

Perhaps the most straightforward isolation goal for Apache/OpenSSL is to protect the server's RSA private key from disclosure to an attacker; holding this key would allow the attacker to recover the session key for any eavesdropped session, past or future. (We presume here that ephemeral, per-connection RSA keys, which

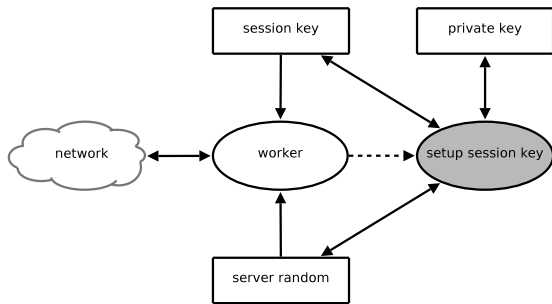


Figure 2: Partitioning to protect against disclosure of private key and arbitrary session key generation.

provide forward secrecy, are not in use—they are rarely used in practice because of their high computational cost.)

We must also prevent an attacker from *using* the RSA private key to decrypt ciphertext of his choosing, and learning the resulting cleartext; such a decryption oracle would serve the attacker equally well for recovering session keys for past or future sessions.

Once the RSA private key is put out of reach for the attacker, how else might he recover a session key that matches one used in an eavesdropped connection? Note that the server *must* include code that generates SSL session keys—if it does not, it cannot complete SSL handshakes. If the attacker can somehow *influence* this code so as to force the server to generate the same session key that was used for a past connection he has eavesdropped, and learn the resulting session key, he will still achieve his goal.

The SSL session key derives from three inputs that traverse the network: random values supplied by the server and client, both sent in clear over the network during the SSL handshake, and another random value supplied by the client, sent over the network encrypted with the server’s public key [15]. Note that by eavesdropping, the attacker learns all three of these values (the last in ciphertext form).

We observe that it is eminently possible to prevent an attacker who exploits the server from usefully influencing the output of the server’s session key generation code. In particular, we may deny the network-facing compartment of the server the privilege to dictate the server’s random contribution to the session key. Instead, a privileged compartment, isolated from the unprivileged one, may supply the server’s random contribution (which is, after all, generated by the server itself).

To meet the above-stated goals, we partitioned Apache 1.3.19 (OpenSSL 0.9.6) as shown in Figure 2. We create one *worker* sthread per connection, which encapsulates unprivileged code. This sthread terminates after serving a single request, to isolate successive requests from one another. We allocate the RSA private key in tagged mem-

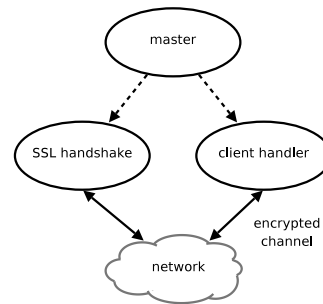


Figure 3: Man-in-the-middle defense: top-level compartmentalization of Apache.

ory. Although the *worker* has no direct access to the private key, it can still complete the SSL handshake and establish a session key via the *setup session key* callgate. This callgate returns the established session key, which does not yield any information regarding the private key. Thus, the only way to obtain the private key is to exploit the callgate itself; we have thus reduced the application’s trusted code base with respect to the private key to the callgate’s contents. In order to prevent an attacker who exploits the *worker* from influencing session key generation, we ensure that the *setup session key* callgate itself generates the *server random* input to session key generation, rather than accepting this input as an argument from the *worker*. Because the session key is a cryptographic hash over three inputs, one of which is random from the attacker’s perspective, he cannot usefully influence the generated session key.

5.1.2 Containing man-in-the-middle attacks

We now consider the man-in-the-middle threat model, wherein the attacker interposes himself between a legitimate client and the server, and can eavesdrop on, forward, and inject messages between them. First, observe that the partitioning described for the previous threat model does not protect a legitimate client’s session key in this stronger model. If the attacker exploits the server’s *worker*, and then passively passes messages as-is between the client and server, then this compromised *worker* will run the attacker’s injected code during the SSL handshake with the legitimate client. The attacker may then allow the legitimate client to complete the SSL handshake, and leak the legitimate client’s session key (readable by the *worker*) to the attacker. The defense against this subtler attack, as one might expect, is finer-grained partitioning of Apache/OpenSSL with Wedge.

Recall that there are two phases to an SSL session. The first phase consists of the SSL handshake and authentication. After this first phase, the client has authenticated the server, by verifying that the server can decrypt random data encrypted with the server’s public key. In addition, the client and server have agreed on a session key to use as the basis for an encrypted and MAC’ed channel

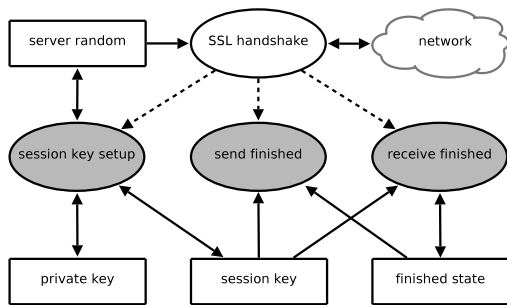


Figure 4: SSL handshake compartmentalization.

between them. In the second phase, application requests and responses traverse this channel.

For privilege separation, there are different threat models for the two phases, so we treat them separately. Figure 3 shows the high-level partitioning needed to implement this two-phase approach. The master exists purely to start and stop two sthreads, and enforce that they only execute sequentially. The *SSL handshake* sthread handles the first phase. This sthread must be able to read and write cleartext to the network to perform the handshake, and while it must be able to call callgates to generate the session key, it should not have access to the resulting key. As this cleartext-reading, network-facing sthread may be exploited, its privileges are limited to those needed for the handshake.

Once the *SSL handshake* sthread completes the handshake, it terminates. The master, which waits for this sthread to terminate, only then starts the *client handler* sthread to handle the second phase. If the attacker attempts to mount a man-in-the-middle attack during the first phase, there are only two possible outcomes: either the handshake fails to complete, and no harm is done, or it does complete, and generates a session key that is not known to the attacker. The *client handler* sthread, however, *does* have access to this session key, and thus it is connected via the SSL-encrypted and -MAC'ed channel to the legitimate client. Despite the man-in-the-middle attack during the first phase, the attacker ends up on the outside of this protected channel.

Man-in-the-middle attacks during the second phase are much more difficult for the attacker, because of the MAC protection on the channel. Data injected by the attacker will be rejected by the *client handler* sthread, and not reach further application code. The attacker's only recourse now is to attempt to exploit the symmetric key decryption code itself. This code is much simpler to audit for vulnerabilities, but as we shall show, further partitioning is also possible to provide defense in depth. We now describe the details of these two phases.

First stage: SSL Handshake Figure 4 shows the partitioning for the *SSL handshake* stage. The *private key*

memory region contains the server's private key, and the *session key* memory region stores the session key. The network-facing *SSL handshake* sthread coordinates the SSL handshake and establishes the session key, *without* being able to read or write it directly; as shown in Figure 4, *SSL handshake* holds neither read nor write permissions for the *session key* tagged memory region. Nevertheless, during the SSL handshake protocol, the server must decrypt one message and encrypt one message with the session key. *SSL handshake* cannot be permitted to invoke callgates that simply encrypt or decrypt their arguments, either; if *SSL handshake* were exploited, the attacker could use them as encryption and decryption oracles, to decrypt ciphertext from the legitimate client.

To understand how one may partition the server to deny *SSL handshake* access to an encryption or decryption oracle for the session key, yet still allow the uses of the session key required by the SSL handshake, one must examine more closely how the SSL handshake protocol uses the session key. After the server and client agree on the session key, they exchange session-key-encrypted SSL Finish messages, to verify that both parties agree on the content of all prior messages exchanged during the handshake. Each SSL Finish message includes a hash derived from all prior messages sent and received by its sender.

We instantiate two callgates, both of which *SSL handshake* may invoke: *receive finished*, which processes the client's SSL Finish, and *send finished*, which generates the server's SSL Finish. *Receive finished* takes a hash derived from all past messages and the client's SSL Finished message as arguments, and must decrypt the SSL Finished message in order to verify it. Once verification is complete, *receive finished* hashes the resulting cleartext together with the hash of past messages, to prepare the payload of the server's SSL Finished message. It stores this result in *finished state*, tagged memory accessible only to the *receive finished* and *send finished* callgates. The only return value seen by *SSL handshake* is a binary success/failure indication for the validation of the client's SSL Finished message. Thus, if *SSL handshake* is exploited, and passes ciphertext from the innocent client to *receive finished* in place of an SSL Finished message, *receive finished* will not reveal the ciphertext.

Send finished simply uses the content of *finished state* to prepare the server's SSL Finished message, and takes no arguments from *SSL handshake*. Data does flow from *SSL handshake* into *finished state*, via *receive finished*. But as *receive finished* hashes this data, an attacker who has exploited *SSL handshake* cannot choose the input that *send finished* encrypts, by the hash function's non-invertibility.

We conclude that if the attacker exploits the *SSL handshake* sthread, he will have no direct access to the session

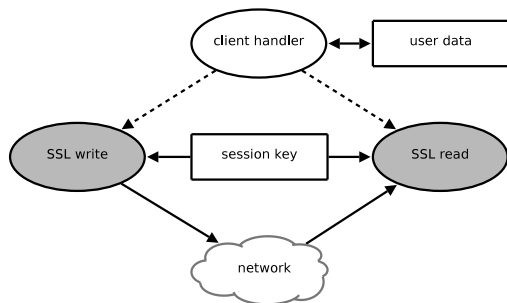


Figure 5: Client handler compartmentalization.

key and, equally important, no access to an encryption or decryption oracle for the session key.

Our implementation fully supports SSL session caching, which varies the final message exchanges in the SSL handshake slightly; we omit those details in the interest of brevity.

Second stage: SSL Client Handler After *SSL handshake* completes the handshake, it exits. The *master*, previously awaiting this event, then starts *client handler* to process the client’s requests. (If *SSL handshake* is exploited, and does not exit, the *master* will not start the *client handler*.)

Figure 5 depicts the compartmentalization of the *client handler* phase. Here we use two callgates, *SSL read* and *SSL write*, to perform encryption and decryption operations using the session key, respectively. Since *client handler* does not have direct access to the network, an attacker must inject correctly MAC’ed and encrypted ciphertext into the user’s connection to compromise *client handler*; otherwise, the injected messages will be dropped by *SSL read* (because the MAC will fail). Because of the partitioning of *SSL handshake*, though, the attacker cannot learn the session key, which includes the MAC key. And thus, the attacker will not be able to exploit *client handler*, and so won’t be able to leak users’ cleartext data (stored in the *user data* memory region).

One unique aspect of the partitioning in Figure 5 is that the unprivileged *client handler* thread does not have write access to the network. This design choice is an instance of defense-in-depth. In the extremely unlikely event that the attacker manages to exploit *SSL read*, he cannot directly leak the session key or user data to the network, as *SSL read* does not have write access to the network. If, however, the attacker next exploits *client handler* by passing it a maliciously constructed return value, he will still have no direct network access. He will only be able to leak sensitive data as ciphertext encrypted by *SSL write*, and thus only over a covert channel, such as by modulating it over the time intervals between sent packets.

Partitioning Metrics A partitioning’s value is greatest when the greatest fraction of code that processes network-derived input executes in shtreads, and the least fraction in callgates. The latter (man-in-the-middle) partitioning of Apache/OpenSSL we’ve described contains $\approx 16\text{K}$ lines of C code that execute in callgates, and $\approx 45\text{K}$ lines of C code that execute in shtreads, including comments and empty lines. As all code in either category ran as privileged originally, this partitioning reduces the quantity of trusted, network-facing code in Apache/OpenSSL by just under two-thirds. To implement the partitioning, we made changes to ≈ 1700 lines of code, which comprise only 0.5% of the total Apache/OpenSSL code base.

We note in closing that we relied heavily on Crowbar during our partitioning of Apache/OpenSSL. For example, enforcing a boundary between Apache/OpenSSL’s worker and master shtreads required identifying 222 heap objects and 389 globals. Missing even one of these results in a protection violation and crash under Wedge’s default-deny model. Crowbar greatly eased identifying these memory regions and their allocation sites, and the shtreads that needed permissions for them.

5.2 OpenSSH

OpenSSH provides an interesting test case for Wedge. Not only was it written by security-conscious programmers, but it is also a leading example of process-level privilege separation [13].

The application-dictated goals for partitioning OpenSSH are:

- Minimize the code with access to the server’s private key.
- Before authentication, run with minimal privilege, so that exploits are contained.
- After authentication, escalate to full privileges for the authenticated user.
- Prevent bypassing of authentication, even if the minimally privileged code is exploited.

When partitioning OpenSSH, we started from scratch with OpenSSH version 3.1p1, the last version prior to the introduction of privilege separation. Clearly, an unprivileged shtread is a natural fit for the network-facing code during authentication. As shtreads do not inherit memory, there is no need to scrub, as when creating a slave with *fork* in conventional privilege separation. We explicitly give the shtread read access to the server’s public key and configuration options, and read/write access to the connection’s file descriptor. We further restrict the shtread by running it as an unprivileged user and setting its filesystem root to an empty directory.

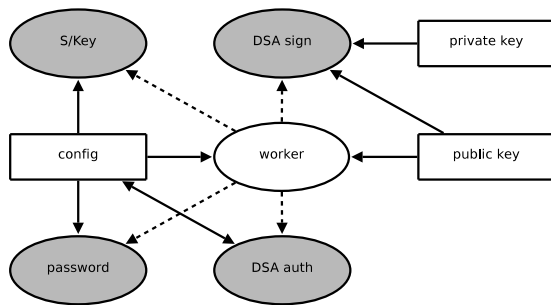


Figure 6: OpenSSH compartmentalization.

The server’s private key is sensitive data, and so must be protected behind a callgate. In our implementation, this callgate contains only 280 lines of C, a small fraction of the total OpenSSH code base, all of which could access the private key in monolithic OpenSSH. To allow authentication, we implemented three more callgates—one each for password, DSA key-based, and S/Key challenge-response authentication. The resulting compartmentalization of OpenSSH appears in Figure 6.

The master sthread (not shown) will spawn one worker for each connection. The worker sthread has no way of tampering with the master, as it cannot write to any memory shared with it. It also cannot tamper with other connections, as sthreads share no memory by default. Thus, all workers (connections) are isolated from each other and from the rest of the OpenSSH daemon.

The worker needs access to the public key in order to reveal its identity to the client. It needs access to the configuration data to supply version strings, supported ciphers, &c. to the client. It has no direct access to sensitive data such as the server’s private key or user credentials, and can only make use of these data via callgates.

When the server authenticates itself to the client, it must sign data using its private key. The *DSA sign* callgate takes a data stream as input, and returns its signed hash. The worker cannot sign arbitrary data, and therefore possibly decrypt data, since only the hash computed by the callgate is signed. Exploiting this callgate is the only way that the worker can obtain the private key.

When the user authenticates himself to the server, the *Password*, *DSA auth* or *S/Key* callgate will be invoked, depending on the authentication mechanism negotiated. The password authentication callgate needs access to the configuration data to check whether the user is allowed to login, whether empty passwords are permitted, and whether password authentication is allowed at all. It also needs access to the shadow password file, which is read directly from disk; it has these privileges because it inherits the filesystem root of its creator, not of its caller. The DSA authentication callgate can determine this information by inspecting the user’s allowed keys in the file system, and the S/Key callgate can by checking whether

the user has an entry in the S/Key database.

The only way for the worker to change its user ID, and thus effectively let the user log in, is for it to successfully authenticate via a callgate. If the authentication is “skipped” by simply not invoking the callgate, the worker will remain unprivileged. The callgate, upon successful authentication, changes the worker’s user ID and filesystem root—an idiom previously applied by Privtrans [1]. Thus, the only way to log in without knowing the user’s credentials is to exploit one of the authentication callgates.

We gleaned two important lessons by comparing the Wedge-partitioned OpenSSH with today’s privilege-separated OpenSSH. The first concerns the importance of avoiding subtle information leaks from a privileged to an unprivileged compartment. Consider password authentication in (non-Wedge) privilege-separated OpenSSH, which proceeds in two steps. First, the unprivileged slave process sends the username to the privileged monitor process, which either returns NULL if that username does not exist, or the *passwd* structure for that username. Second, the slave sends the password to the monitor, which authenticates the user. The result of the first interaction with the monitor is in fact an information leak—it would allow an exploited slave to invoke the monitor at will to search for valid usernames. This vulnerability remains in today’s portable OpenSSH 4.7. The Wedge partitioning keeps OpenSSH’s two-step authentication for ease of coding reasons, but the *password* callgate in Figure 6 returns a dummy *passwd* struct (rather than NULL) when the username doesn’t exist; this way, even an exploited worker cannot use the *password* callgate to search for usernames. A prior version of OpenSSH contained a similar vulnerability, wherein an S/Key challenge would only be returned if a valid username had been supplied by the remote client [14]. In this case, the OpenSSH monitor and slave leak sensitive information directly to the network; an attacker needn’t exploit the slave.

The second lesson concerns the value of default-deny permissions. A past version of OpenSSH suffered from a vulnerability in which the PAM library (not written by the OpenSSH authors, but called by OpenSSH) kept sensitive information in scratch storage, and did not scrub that storage before returning [8]. If a slave that inherited this memory via *fork* were exploited, it could disclose this data to an attacker. A PAM callgate would not be subject to this vulnerability; any scratch storage allocated with *malloc* within the callgate would be inaccessible by the *worker*.

Partitioning Metrics In the Wedge-partitioned version of OpenSSH, ≈ 3300 lines of C (including comments and whitespace) execute in callgates, and $\approx 14K$ execute in sthreads; as all of these lines of code would

have executed in a single, privileged compartment in monolithic OpenSSH, partitioning with Wedge has reduced the quantity of privileged code by over 75%. Achieving this isolation benefit required changes to 564 lines of code, only 2% of the total OpenSSH code base.

6 Performance

In evaluating Wedge's performance, we have three chief aims. First, we validate that Wedge's isolation primitives incur similar costs to those of the isolation and concurrency primitives commonly used in UNIX. Second, we assess whether applications instrumented with the Crowbar development tool generate traces tolerably quickly. Finally, we quantify the performance penalty that fine-grained partitioning with Wedge's primitives incurs for Apache/OpenSSL's throughput and OpenSSH's interactive latency.

All experiments ran on an eight-core 2.66 GHz Intel Xeon machine with 4 GB of RAM, apart from the Apache experiments, which ran on a single-core 2.2GHz AMD Opteron machine with 2 GB of RAM, to ease saturation.

Wedge primitives: Microbenchmarks To examine the cost of creating and running an sthread, we measured the time elapsed between requesting the creation of an sthread whose code immediately calls *exit* and the continuation of execution in the sthread's parent. This interval includes the time spent on the kernel trap for the sthread creation system call; creating the new sthread's data structures, scheduling the new sthread, executing *exit* in the new sthread, destroying the sthread, and rescheduling the parent sthread (whose timer then stops). We implemented analogous measurements for pthreads, recycled callgates, stthreads, standard callgates, and *fork*. In all these experiments, the originating process was of minimal size.

Figure 7 compares the latencies of these primitives. Stthreads and callgates are of similar cost to *fork*, and recycled callgates are of similar cost to pthread creation. Thus, overall, Wedge's isolation primitives incur similar overhead to familiar isolation and concurrency primitives, but support finer-grained control over privilege.

Callgates perform almost identically to stthreads because they are implemented as separate stthreads. Recycled callgates outperform callgates by a factor of eight, as they reuse an stthread, and thus save the cost of creating a new stthread per callgate invocation. For parents with small address spaces, stthread creation involves similar overhead to that of *fork*. For parents with large page tables, however, we expect stthread creation to be faster than *fork*, because only those entries of the page table and those file descriptors specified in the security policy are copied for a new stthread; *fork* must always copy these

in their entirety. Stthreads are approximately 8x slower than pthreads, which incur minimal creation cost (no resource copying) and low context switch overhead (no TLB flush).

Figure 8 shows the cost of creating tags. Allocating memory from a tagged region using *smalloc* costs roughly the same as standard *malloc*, as the two allocators are substantially the same. The difference in overhead between the two lies in tag creation, which is essentially an *mmap* operation, followed by initialization of *malloc* data structures. We manage to outperform *mmap* by caching and reusing previously deleted tags, as described in Section 4.1. The *tag_new* result shown considers the best case, where reuse is always possible; in this case, the operation is approximately 4x slower than *malloc*. In the worst case, when no reuse is possible, tag creation costs similarly to *mmap*, and is hence 22x slower than *malloc*. We expect reuse to be common in network applications, as the master typically creates tags on a per-client basis, so new client stthreads can benefit from completing ones. Indeed, this mechanism improved the throughput of our partitioned Apache server by 20%.

Crowbar: Run-time Overhead Figure 9 shows the elapsed time required to run OpenSSH, Apache, and most of the C-language SPECint2006 benchmark applications under *cb-log* (we omit three of these from the figure in the interest of brevity, as they performed similarly to others). We compare these elapsed times against those required to run each application under Pin with *no* instrumentation (*i.e.*, the added cost of Pin alone), and those required to run the "native" version of each application, without Pin. We do not report performance figures for *cb-analyze*, as it consistently completes in seconds. All experiments were conducted using Pin version 2.3.

All applications we measured under *cb-log* (SPEC benchmarks, OpenSSH, and Apache) produced traces in less than ten minutes, and in a mean time of 76 seconds. For the range of applications we examined, *cb-log*'s instrumentation overhead is tolerable. Trace generation occurs only during the development process, and a single trace reveals much of the memory access behavior needed to partition the application. Absolute completion time is more important than relative slowdown for a development tool. Indeed, obtaining a trace from OpenSSH incurs an average 46x slowdown vs. native OpenSSH, (2.4x vs. Pin without instrumentation), yet the trace for a single login takes less than four seconds to generate.

Pin instruments each fetched basic block of a program once, and thereafter runs the cached instrumented version. From Figure 9, we see that Pin on average executes the applications we measured approximately 7x slower than they execute natively. Note that Pin's overhead is least for applications that execute basic blocks

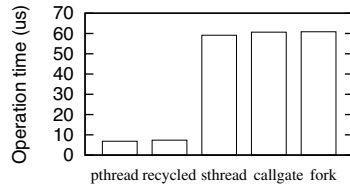


Figure 7: Sthread calls.

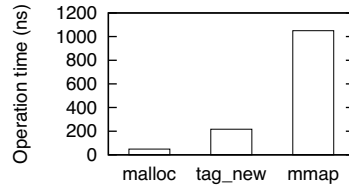


Figure 8: Memory calls.

Experiment	Vanilla	Wedge	Recycled
Apache sessions cached (req/s)	1238	238	339
Apache sessions not cached (req/s)	247	132	170
ssh login delay (s)	0.145	0.148	
10MB scp delay (s)	0.376	0.370	

Table 2: OpenSSH and Apache performance.

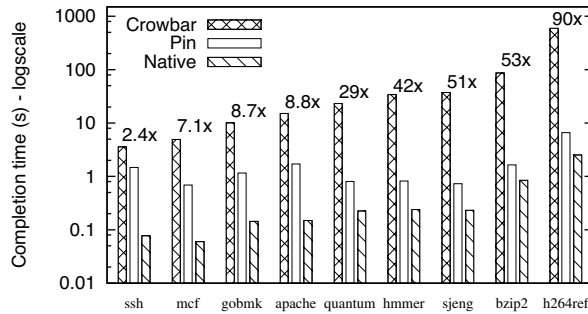


Figure 9: Overhead of *cb-log*. The number above an application’s bars indicates the ratio between run time under Pin without instrumentation and the run time under Pin with Crowbar.

many times, as do many of the SPEC benchmarks. For OpenSSH and Apache, which do not repeatedly execute basic blocks to the extent the SPEC benchmarks do, caching of instrumented code pays less of a dividend.

Cb-log must of course execute added instrumentation instructions in each basic block, and hence it slows applications much more than Pin alone does. On average, applications run 96x more slowly under *cb-log* than they do natively, and 27x more slowly under *cb-log* than they do under Pin with no instrumentation.

Applications: End-to-end Performance The top half of Table 2 shows the maximum throughput that the Wedge-partitioned version of Apache and the original version of Apache can sustain, in requests per second. We consider two Wedge-partitioned implementations: one with standard callgates (“Wedge”), and one with recycled callgates (“Recycled”). We also measured Apache’s performance with and without session caching. In these experiments, four client machines request static web pages using SSL, across a 1 Gbps Ethernet.

The isolation benefits Wedge offers Apache come at a performance cost that depends on the client workload. When all client SSL sessions are cached (and thus, the server need not execute the cryptographic operations in the SSL handshake), Wedge-partitioned Apache with recycled sthreads achieves only 27% of the throughput of unpartitioned Apache. When no SSL sessions are cached, however, Wedge-partitioned Apache with recycled sthreads reaches 69% of unpartitioned Apache’s throughput. The all-sessions-cached workload is entirely

unrealistic; we include it only to show the worst-case overhead that partitioning Apache with Wedge incurs.

Because SSL session caching allows clients to reuse session keys, it eliminates the significant SSL handshake cost that the server incurs once per connection in the non-session-cached workload. Thus, Wedge-induced overhead is a greater fraction of the total computation required when the server accepts a connection for a cached session. This overhead is directly proportional to the number of sthreads and callgates created and invoked per request. Each request creates two sthreads and invokes eight callgates (nine, for non-session-cached clients)—a few callgates are invoked more than once per request. Vanilla Apache instead uses a pool of (reused) workers, so it does not pay process creation overhead on each request; it thus provides no isolation between successive requests executed by the same worker, but is faster. Recycled callgates are an effective optimization—they increase Wedge-partitioned Apache’s throughput by 42% and 29% for workloads with and without session caching, respectively.

The bottom half of Table 2 compares the latencies of operations in pre-privilege-separated OpenSSH and in the Wedge-partitioned version, for a single OpenSSH login and for uploading a single 10 MB file using scp. Wedge’s primitives introduce negligible latency into the interactive OpenSSH application.

7 Discussion

We now briefly discuss a few of Wedge’s limitations, and topics that bear further investigation. Several of these limitations concern callgates. First, we rely on their not being exploitable. Second, the interface to a callgate must not leak sensitive data: neither through its return value, nor through any side channel. If a return value from a callgate does so, then the protection within the callgate is of no benefit. More subtly, callgates that return values derived from sensitive data should be designed with great care, as they may be used by an adversary who can exploit an unprivileged caller of the callgate either to derive the sensitive data, or as an oracle, to compute using the sensitive data without being able to read it directly.

We trust the kernel support code for sthreads and callgates. As this code is of manageable size—less than 2000

lines—we believe that it can be audited. Perhaps more worryingly, we must also trust the remainder of the Linux kernel, though like Flume [7], we also inherit Linux’s support for evolving hardware platforms and compatibility with legacy application code.

Crowbar is an aid to the programmer; not a tool that automatically determines a partitioning by taking security decisions on its own. We believe that automation could lead to subtle vulnerabilities, such as those described in Section 5.1.2, that are not apparent from data dependencies alone. Similarly, one must use Crowbar with caution. That is, one must assess which permissions should be granted to an sthread, and which need to be wrapped around a callgate. The tool alone guarantees no security properties; it merely responds to programmer queries as a programming aid, and it is the programmer who enforces correct isolation.

Wedge does not deny read access to the text segment; thus, a programmer cannot use the current Wedge implementation to prevent the *code* for an sthread (or indeed, its ancestors) from being leaked. Wedge provides no direct mechanism to prevent DoS attacks, either; an exploited sthread may maliciously consume CPU and memory.

We intend to explore static analysis as an alternative to runtime analysis. Static analysis will yield a superset of the required permissions for an sthread, as some code paths may never execute in practice. Static analysis would report the exhaustive set of permissions for an sthread not to encounter a protection violation. Yet these permissions could well include privileges for sensitive data that could allow an exploit to leak that data. By using run-time analysis of the application running on an innocuous workload, the programmer learns which privileges are used when an exploit does *not* occur, but only those required for correct execution for *that workload*.

8 Related Work

Many have recognized the practical difficulty of partitioning applications in accordance with least privilege. OKWS [5] showed that with appropriate contortions, UNIX’s process primitives can be used to build a web server of compartments with limited privileges. Krohn *et al.* [6] lament that UNIX is more of an opponent than an ally in the undertaking, with the result that programmers create single-compartment, monolithic designs, wherein all code executes with the union of all privilege required across the entire application.

A trio of Decentralized Information Flow Control (DIFC) systems, Asbestos, HiStar, and Flume [2, 7, 18] offer a particularly general approach to fine-grained isolation. DIFC allows untrusted code to observe sensitive data, but without sufficient privilege to disclose that data. Wedge does not provide any such guarantees for

untrusted code. In its all-or-nothing privilege model, a compartment is either privileged or unprivileged with respect to a resource, and if a privileged compartment is exploited, the attacker controls that compartment’s resources with that compartment’s privileges. Our experience thus far suggests that when applying Wedge to legacy, monolithic code, only a small fraction of the original code need run in privileged compartments. The price of DIFC is heightened concern over covert channels, and mechanisms the programmer must employ to attempt to eliminate them [18]—once one allows untrusted code to see sensitive data, one must restrict that code’s communication with the outside world. Because Wedge is not a DIFC system, it does not share this characteristic. Wedge still may disclose sensitive information through covert channels, but *only* from compartments that are privileged with respect to sensitive information. DIFC systems leave the daunting challenge of designing an effective partitioning entirely to the programmer. Wedge’s Crowbar tools focus largely on helping him solve this complementary problem; we believe that similar tools may prove useful when partitioning applications for Asbestos, HiStar, and Flume, as well.

Jif [12] brings fine-grained control of information flow to the Java programming language. Its decentralized label model directly inspired DIFC primitives for operating systems. Jif uses static analysis to track the propagation of labels on data through a program, and validate whether the program complies with a programmer-supplied information flow control policy. The Wedge isolation primitives do not track the propagation of sensitive data; they use memory tags only to allow the programmer to grant privileges across compartment boundaries. Wedge’s Crowbar development tool, however, provides simple, “single-hop” tracking of which functions use which memory objects in legacy C code, whose lack of strong typing and heavy use of pointers make accurate static analysis a challenge. Jif uses static analysis to detect *prohibited* information flows, while Crowbar uses dynamic analysis to reveal to the programmer what are most often *allowed* information flows, to ease development for Wedge’s default-deny model.

Jif/split [17] extends labeled information flow for Java to allow automated partitioning of a program across distributed hosts, while preserving confidentiality and integrity across the entire resulting ensemble. Because it allows principals to express which hosts they trust, Jif/split can partition a program such that a code fragment that executes on a host only has access to data owned by principals who trust that host. Wedge targets finer-grained partitioning of an application on a single host. In cases where robustness to information leakage depends on the detailed semantics of a cryptographic protocol, such as in the defense against man-in-the-middle attacks on SSL in

Section 5.1, Jif/split would require careful, manual placement of declassification, based on the same programmer knowledge of the protocol's detailed semantics required when using Wedge's primitives.

Provos proposes privilege separation [13], a special case of least-privilege partitioning targeted specifically for user authentication, in which a monolithic process is split into a privileged monitor and one (or more) unprivileged slaves, which request one of a few fixed operations from the monitor using IPC. Privman [4] is a reusable library for use in implementing privilege-separated applications. Privilege separation allows sharing of state between the monitor and slave(s), but does not assist the programmer in determining what to share; Crowbar addresses this complementary problem. Privtrans [1] uses programmer annotations of sensitive data in a server's source code and static analysis to automatically derive a two-process, privilege-separated version of that server. Wedge arguably requires more programmer effort to use than Privtrans, but also allows much richer partitioning and tighter permissions than these past privilege separation schemes; any number of sthreads and callgates may exist within an application, interconnected in whatever pattern the programmer specifies, and they may share disjoint memory regions with one another at a single-byte granularity.

9 Conclusion

Programmers know that monolithic network applications are vulnerable to leakage or corruption of sensitive information by bugs and remote exploits. Yet they still persist in writing them, because they are far easier to write than carefully least-privilege-partitioned ones. Introducing tightly privileged compartments into a legacy monolithic server is even harder, because memory permissions are implicit in monolithic code, and even a brief fragment of monolithic code often uses numerous scattered memory regions. The Wedge system represents our attempt to exploit the synergy between simple, default-deny isolation primitives on the one hand, and tools to help the programmer reason about the design and implementation of the partitioning, on the other. Our experience applying Wedge to Apache/OpenSSL and OpenSSH suggests that it supports tightly privileged partitioning of legacy networking applications well, at acceptable performance cost. Above all, we are particularly encouraged that Wedge has proven flexible and powerful enough to protect Apache against not only simple key leakage, but also against complex man-in-the-middle attacks.

Wedge is publicly available at:

<http://nrg.cs.ucl.ac.uk/wedge/>

Acknowledgments

We thank our shepherd Eddie Kohler and the anonymous reviewers for their insightful comments, and David Mazières and Robert Morris for illuminating discussions during the course of this work.

Mark Handley and Brad Karp are supported by Royal Society-Wolfson Research Merit Awards. Karp is further supported by funding from Intel Corporation.

References

- [1] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, 2004.
- [2] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.
- [3] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furewicks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.
- [4] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Security*, *FREENIX Track*, 2003.
- [5] M. Krohn. Building secure high-performance web services with OKWS. In *USENIX*, Boston, MA, June 2004.
- [6] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *HotOS*, 2005.
- [7] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.
- [8] M. Kuhn. OpenSSH PAM conversation memory scrubbing weakness, 2003. <http://www.securityfocus.com/bid/9040>.
- [9] D. Lea. A memory allocator by Doug Lea. <http://g.oswego.edu/dl/html/malloc.html>.
- [10] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX*, 2001.
- [11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
- [12] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–442, 2000.
- [13] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.
- [14] Rembrandt. OpenSSH S/Key remote information disclosure vulnerability, 2002. <http://www.securityfocus.com/bid/23601>.
- [15] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000.
- [16] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [17] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, 2001.
- [18] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2007.

Reducing Network Energy Consumption via Sleeping and Rate-Adaptation

Sergiu Nedevschi^{*†} Lucian Popa^{*†} Gianluca Iannaccone[†]
Sylvia Ratnasamy[†] David Wetherall^{‡§}

Abstract

We present the design and evaluation of two forms of power management schemes that reduce the energy consumption of networks. The first is based on putting network components to sleep during idle times, reducing energy consumed in the absence of packets. The second is based on adapting the rate of network operation to the offered workload, reducing the energy consumed when actively processing packets.

For real-world traffic workloads and topologies and using power constants drawn from existing network equipment, we show that even simple schemes for sleeping or rate-adaptation can offer substantial savings. For instance, our practical algorithms stand to halve energy consumption for lightly utilized networks (10-20%). We show that these savings approach the maximum achievable by any algorithms using the same power management primitives. Moreover this energy can be saved without noticeably increasing loss and with a small and controlled increase in latency ($<10\text{ms}$). Finally, we show that both sleeping and rate adaptation are valuable depending (primarily) on the power profile of network equipment and the utilization of the network itself.

1 Introduction

In this paper, we consider power management for networks from a perspective that has recently begun to receive attention: the conservation of energy for operating and environmental reasons. Energy consumption in network exchanges is rising as higher capacity network equipment becomes more power-hungry and requires greater amounts of cooling. Combined with rising energy costs, this has made the cost of powering network exchanges a substantial and growing fraction of the total cost of ownership – up to half by some estimates[23]. Various studies now estimate the power usage of the US network infrastructure at between 5 and 24 TWh/year[25, 26], or \$0.5-2.4B/year at a rate of \$0.10/KWh, depending on what is included. Public concern about carbon footprints is also rising, and stands to affect network equipment much as it has computers

via standards such as EnergyStar. In fact, EnergyStar standard proposals for 2009 discuss slower operation of network links to conserve energy when idle. A new IEEE 802.3az Task Force was launched in early 2007 to focus on this issue for Ethernet [15].

Fortunately, there is an opportunity for substantial reductions in the energy consumption of existing networks due to two factors. First, networks are provisioned for worst-case or busy-hour load, and this load typically exceeds their long-term utilization by a wide margin. For example, measurements reveal backbone utilizations under 30% [16] and up to hour-long idle times at access points in enterprise wireless networks [17]. Second, the energy consumption of network equipment remains substantial even when the network is idle. The implication of these factors is that *most* of the energy consumed in networks is wasted.

Our work is an initial exploration of how overall network energy consumption might be reduced without adversely affecting network performance. This will require two steps. First, network equipment ranging from routers to switches and NICs will need power management primitives at the hardware level. By analogy, power management in computers has evolved around hardware support for *sleep* and *performance* states. The former (*e.g.*, C-states in Intel processors) reduce idle consumption by powering off sub-components to different extents, while the latter (*e.g.*, SpeedStep, P-states in Intel processors) tradeoff performance for power via operating frequency. Second, network protocols will need to make use of the hardware primitives to best effect. Again, by analogy with computers, power management preferences control how the system switches between the available states to save energy with minimal impact on users.

Of these two steps, our focus is on the network protocols. Admittedly, these protocols build on hardware support for power management that is in its infancy for networking equipment. Yet the necessary support will readily be deployed in networks where it proves valuable, with forms such as sleeping and rapid rate selection for Ethernet [15] already under development. For comparison, computer power management compatible with the ACPI standard has gone from scarce to widely deployed over the past five to ten years and is now expanding into the server market. Thus our goal is

^{*}University of California, Berkeley

[†]Intel Research, Berkeley

[‡]University of Washington

[§]Intel Research, Seattle

to learn: what magnitude of energy savings a protocol using feasible hardware primitives might offer; what performance tradeoff comes with these savings; and which of the feasible kinds of hardware primitives would maximize benefits. We hope that our research can positively influence the hardware support offered by industry.

The hardware support we assume from network equipment is in the form of performance and sleep states. Performance states help to save power when routers are active, while sleep states help to save power when routers are idle. The performance states we assume dynamically change the rate of links and their associated interfaces. The sleep states we assume quickly power off network interfaces when they are idle. We develop two approaches to save energy with these primitives. The first puts network interfaces to sleep during short idle periods. To make this effective we introduce small amounts of buffering, much as 802.11 APs do for sleeping clients; this collects packets into small bursts and thereby creates gaps long enough to profitably sleep. Potential concerns are that buffering will add too much delay across the network and that bursts will exacerbate loss. Our algorithms arrange for routers and switches to sleep in a manner that ensures the buffering delay penalty is paid only once (not per link) and that routers clear bursts so as to not amplify loss noticeably. The result is a novel scheme that differs from 802.11 schemes in that all network elements are able to sleep when not utilized yet added delay is bounded. The second approach adapts the rate of individual links based on the utilization and queuing delay of the link.

We then evaluate these approaches using real-world network topologies and traffic workloads from Abilene and Intel. We find that: (1) rate-adaptation and sleeping have the potential to deliver substantial energy savings for typical networks; (2) the simple schemes we develop are able to capture most of this energy-saving potential; (3) our schemes do not noticeably degrade network performance; and (4) both sleeping and rate-adaptation are valuable depending primarily on the utilization of the network and equipment power profiles.

2 Approach

This section describes the high-level model for power consumption that motivates our rate adaptation and sleeping solutions, as well as the methodology by which we evaluate these solutions.

2.1 Power Model Overview

Active and idle power A network element is *active* when it is actively processing incoming or outgoing traffic, and *idle* when it is powered on but does not process traffic. Given these modes, the energy consumption for

a network element is:

$$E = p_a T_a + p_i T_i \quad (1)$$

where p_a , p_i denote the power consumption in active and idle modes respectively and T_a , T_i the times spent in each mode.

Reducing power through sleep and performance states Sleep states lower power consumption by putting sub-components of the overall system to sleep when there is no work to process. Thus sleeping reduces the power consumed when idle, *i.e.*, it reduces the $p_i T_i$ term of Eqn. (2) by reducing the p_i to some sleep-mode power draw p_s where $p_s < p_i$.

Performance states reduce power consumption by lowering the rate at which work is processed. As we elaborate on in later sections, some portion of both active and idle power consumption depends on the frequency and voltage at which work is processed. Hence performance states that scale frequency and/or voltage reduce both the p_a and p_i power draws resulting in an overall reduction in energy consumption.

We also assume a penalty for transitioning between power states. For simplicity, we measure this penalty in time, typically milliseconds, treating it as a period in which the router can do no useful work. We use this as a simple switching model that lumps all penalties, ignoring other effects that may be associated with switches such as a transient increase in power consumption. Thus there is also a cost for switching between states.

Networks with rate adaptation and sleeping support

In a network context, the sleeping and rate adaptation decisions one router makes fundamentally impacts – and is impacted by – the decisions of its neighboring routers. Moreover, as we see later in the paper, the strategies by which each is best exploited are very different (Intuitively this is because sleep-mode savings are best exploited by maximizing idle times, which implies processing work as quickly as possible, while performance-scaling is best exploited by processing work as slowly as possible, which reduces idle times). Hence, to avoid complex interactions, we consider that the whole network, or at least well-defined components of it, run in either rate adaptation or sleep mode.

We develop the specifics of our sleeping schemes in Section 3, and our rate adaptation schemes in Section 4. Note that our solutions are deliberately constructed to apply *broadly* to the networking infrastructure – from end-host NICs, to switches, and IP routers, etc. – so that they may be applied wherever they prove to be the most valuable. They are not tied to IP-layer protocols.

2.2 Methodology

The overall energy savings we can expect will depend on the extent to which our power-management algorithms can successfully exploit opportunities to sleep or rate adapt as well as the power profile of network equipment (*i.e.*, relative magnitudes of p_a , p_i and p_s). To clearly separate the effect of each, we evaluate sleep solutions in terms of the fraction of time for which network elements can sleep and rate-adaptation solutions in terms of the reduction in the average rate at which the network operates.¹ In this way we assess each solution with the appropriate baseline. We then evaluate how these metrics translate into overall network energy savings for different equipment power profiles and hence compare the relative merits of sleeping and rate-adaptation (Section 5). For both sleep and rate-adaptation, we calibrate the savings achieved by our practical solutions by comparing to the maximum savings achievable by optimal, but not necessarily practical, solutions.

In network environments where packet arrival rates can be highly non-uniform, allowing network elements to transition between operating rates or sleep/active modes with corresponding transition times can introduce additional packet delay, or even loss, that would have not otherwise occurred. Our goal is to explore solutions that usefully navigate the tradeoff between potential energy savings and performance. In terms of performance, we measure the average and 98th percentile of the end-to-end packet delay and loss.

In the absence of network equipment with hardware support for power management, we base our evaluations on packet-level simulation with real-world network topologies and traffic workloads. The key factors on which power savings then depend, beyond the details of the solutions themselves, are the technology constants of the sleep and performance states and the characteristics of the network. In particular, the utilization of links determines the relative magnitudes of T_{active} and T_{idle} as well as the opportunities for profitably exploiting sleep and performance states. We give simple models for technology constants in the following sections. To capture the effect of the network on power savings, we drive our simulation with two realistic network topologies and traffic workloads (Abilene and Intel) that are summarized below. We use *ns2* as our packet-level simulator.

Abilene We use Abilene as a test case because of the ready availability of detailed topology and traffic information. The information from [27] provides us with the link connectivity, weights (to compute routes), latencies and capacities for Abilene’s router-level topology. We use measured Abilene traffic matrices (TMs) available in the community [29] to generate realistic workloads over this topology. Unless otherwise stated, we use as

our default a traffic matrix whose link utilization levels reflect the average link utilization over the entire day – this corresponds to a 5% link utilization on average with bottleneck links experiencing about 15% utilization.

We linearly scale TMs to study performance with increasing utilization up to a maximum average network utilization of 31% as beyond this some links reach very high utilizations. Finally, while the TMs specify the 5-minute average rate observed for each ingress-egress pair, we still require a packet-level traffic generation model that creates this rate. In keeping with previous studies [18, 31] we generate traffic as a mix of Pareto flows, and for some results we use constant bit-rate (CBR) traffic. As per standard practice, we set router queue sizes equal to the bandwidth-delay product in the network; we use the bandwidth of the bottleneck link, and a delay of 100ms.

Intel As an additional real-world dataset, we collected topology and traffic information for the global Intel enterprise network. This network connects Intel sites worldwide, from small remote offices to large multi-building sites with thousands of users. It comprises approximately 300 routers and over 600 links with capacities ranging from 1.5Mbps to 1Gbps.

To simulate realistic traffic, we collected unsampled Netflow records [7] from the core routers. The records, exported by each router every minute, contain per flow information that allows us to recreate the traffic sourced by ingress nodes.

3 Putting Network Elements to Sleep

In this section we discuss power management algorithms that exploit sleep states to reduce power consumption during idle times.

3.1 Model and Assumptions

Background A well established technique, as used by microprocessors and mobiles, is to reduce idle power by putting hardware sub-components to sleep. For example, modern Intel processors such as the Core Duo [1] have a succession of sleep states (called C-states) that offer increasingly reduced power at the cost of increasingly high latencies to enter and exit these states. We assume similar sleep states made available for network equipment. For the purpose of this study we ignore the options afforded by multiple sleep states and assume as an initial simplification that we have a single sleep state.

Model We model a network sleep state as characterized by three features or parameters. The first is the power draw in sleep mode p_s which we assume to be a small fraction of the idle mode power draw p_i .

The second characterizing parameter of a sleep state is the time δ it takes to transition in and out of sleep states. Higher values of δ raise the bar on when the network

element can profitably enter sleep mode and hence δ critically affects potential savings. While network interface cards can make physical-layer transitions in as low as $10\mu s$, transition times that involve restoring state at higher layers (memory, operating system) are likely to be higher [13]. We thus evaluate our solutions over a wide range values of transition times.

Finally, network equipment must support a mechanism for invoking and exiting sleep states. The option that makes the fewest assumptions about the sophistication of hardware support is *timer-driven sleeping*, in which the network element enters and exits sleep at well-defined times. Prior to entering sleep the network element specifies the time in the future at which it will exit sleep and all packets that arrive at a sleeping interface are lost. The second possibility, described in [12], is for routers to wake up automatically on sensing incoming traffic on their input ports. To achieve this “wake-on-arrival” (WoA), the circuitry that senses packets on a line is left powered on even in sleep mode. While support for WoA is not common in either computers or interfaces today, this is a form of hardware support that might prove desirable for future network equipment and is currently under discussion in the IEEE 802.3az Task Force [13]. Note that even with wake-on-arrival, bits arriving during the transition period δ are effectively lost. To handle this, the authors in [6] propose the use of “dummy” packets to rouse a sleeping neighbor. A node A that wishes to wake B first sends B a dummy packet, and then waits for time δ before transmitting the actual data traffic. The solutions we develop in this paper apply seamlessly to either timer-driven or WoA-based hardware.

Measuring savings and performance In this section, we measure savings in terms of the percentage of time network elements spend asleep and performance in terms of the average and 98th percentile of the end-to-end packet delay and loss. We assume that individual line cards in a network element can be independently put to sleep. This allows for more opportunities to sleep than if one were to require that a router sleep in its entirety (as the latter is only possible when there is no incoming traffic at any of the incoming interfaces). Correspondingly our energy savings are with respect to interface cards which typically represent a major portion of the overall consumption of a network device. That said, one could in addition put the route processor and switch fabric to sleep at times when all line cards are asleep.

3.2 Approaches and Potential savings

For interfaces that support wake-on-arrival, one approach to exploiting sleep states is that of *opportunistic sleeping* in which link interfaces sleep when idle – i.e., a router is awakened by an incoming (dummy) packet and, after forwarding it on, returns to sleep if no subsequent

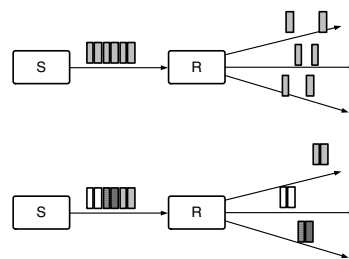


Figure 1: Packets within a burst are organized by destination.

packet arrives for some time. While very simple, such an approach can result in frequent transitions which limits savings for higher transition times and/or higher link speeds. For example, with a 10Gbps link, even under low utilization (5%) and packet sizes of 1KB, the average packet inter-arrival time is very small – $15\mu s$. Thus while opportunistic sleeping might be effective in LANs [11, 21] with high idle times, for fast links this technique is only effective for very low transition times δ (we quantify this shortly). In addition, opportunistic sleep is only possible with the more sophisticated hardware support of wake-on-arrival.

To create greater opportunities for sleep, we consider a novel approach that allows us to explicitly control the tradeoff between network performance and energy savings. Our approach is to shape traffic into small bursts at the edges of the network – edge devices then transmit packets in bunches and routers within the network wake up to process a burst of packets, and then sleep until the next burst arrives. The intent is to provide sufficient bunching to create opportunities for sleep if the load is low, yet not add excessive delay. This is a radical approach in the sense that much other work seeks to avoid bursts rather than create them (e.g., token buckets for QOS, congestion avoidance, buffering at routers). As our measurements of loss and delay show, our schemes avoid the pitfalls associated with bursts because we introduce only a bounded and small amount of burstiness and a router never enters sleep until it has cleared all bursts it has built up. More precisely, we introduce a buffer interval “ B ” that controls the tradeoff between savings and performance. An ingress router buffers incoming traffic for up to B ms and, once every B ms, forwards buffered traffic in a burst.

To ensure that bursts created at the ingress are retained as they traverse through the network, an ingress router arranges packets within the burst such that all packets destined for the same egress router are contiguous within the burst (see figure 1).

The above “buffer-and-burst” approach (B&B) creates alternating periods of contiguous activity and sleep leading to fewer transitions and amortizing the transition penalty δ over multiple packets. This improvement comes at the cost of an added end-to-end delay of up

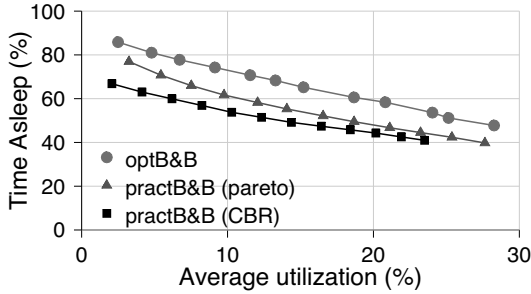


Figure 4: Time asleep using CBR and Pareto traffic.

`practB&B` is simple – it requires no inter-router coordination as the time at which bursts are transmitted is decided independently by each ingress router. For networks supporting wake-on-arrival the implementation is trivial – the only additional feature is the implementation of buffer-and-burst at the ingress nodes. For networks that employ timer-driven sleeping, packet bursts would need to include a marker denoting the end of burst and notifying the router of when it should expect the next burst on that interface.

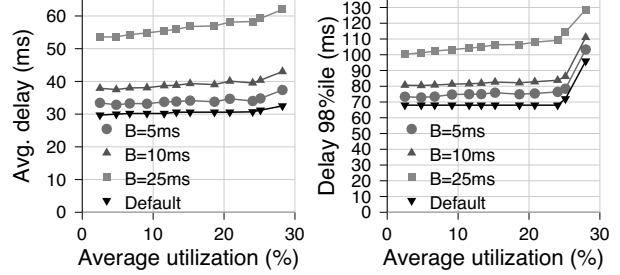
3.4 Evaluation

We evaluate the savings vs. performance tradeoff achieved by `practB&B` algorithm and the impact of equipment and network parameters on the same.

Savings vs. performance using `practB&B` We compare the sleep time achieved by `practB&B` to that achievable by `optB&B`. In terms of performance, we compare the end-to-end packet delay and loss in a network using `practB&B` to that of a network that never sleeps (as today) as this shows the overall performance penalty due to our sleep protocols.

Figure 4 plots the sleep time with increasing utilization on the Abilene network using a buffering interval $B = 10ms$. We plot the percentage sleep time under both CBR and Pareto traffic workloads. We see that even a scheme as simple as `practB&B` can create and exploit significant opportunities for sleep and approaches the savings achieved by the significantly more complex `optB&B`. As with opportunistic sleeping, we see that `practB&B`'s savings with CBR traffic are lower than for the more bursty Pareto workloads, but that this reduction is significantly smaller in the case of `practB&B` than with opportunistic sleeping (recall Figure 3). That Pareto traffic improves savings is to be expected as burstier traffic only enhances our bunching strategy.

Figures 5(a) and (b) plot the corresponding average and 98th percentile of the end-to-end delay. As expected, we see that the additional delay in both cases is proportional to the buffering interval B . Note that this is the end-to-end delay, reinforcing that the buffering delay B is incurred once for the entire end-to-end path.



(a) Average delay (`practB&B`) (b) Delay 98%ile

Figure 5: The impact on delay of `practB&B`.

We see that for higher B , the delay grows slightly faster with utilization (*e.g.*, compare the absolute increase in delay for $B = 5ms$ to $25ms$) because this situation is more prone to larger bursts overlapping at intermediate routers. However this effect is relatively small even in a situation combining larger B ($25ms$) and larger utilizations (20-30%) and is negligible for smaller B and/or more typical utilizations.

We see that both average and maximum delays increase abruptly beyond network utilizations exceeding 25%. This occurs when certain links approach full utilization and queuing delays increase (recall that the utilization on the horizontal axis is the average network-wide utilization). However this increase occurs even in the default network scenario and is thus not caused by `practB&B`'s traffic shaping.

Finally, our measurements revealed that `practB&B` introduced no additional packet loss (relative to the default network scenario) until we approach utilizations that come close to saturating some links. For example, in a network scenario losses greater than 0.1% occur at 41% utilization without any buffering, they occur at 38% utilization with $B = 10ms$, and at 36% utilization with $B = 25$. As networks do not typically operate with links close to saturation point, we do not expect this additional loss to be a problem in practice.

In summary, the above results suggest that `practB&B` can yield significant savings with a very small (and controllable) impact on network delay and loss. For example, at a utilization of 10%, a buffering time of just $B = 5ms$ allows the network to spend over 60% of its time in sleep mode for under $5ms$ added delay.

Impact of hardware characteristics We now evaluate how the transition time δ affects the performance of `practB&B`. Figure 6(a) plots the sleep time achieved by `practB&B` for a range of transition times and compares this to the ideal case of having instantaneous transitions. As expected, the ability to sleep degrades drastically with increasing δ . This observation holds across various buffer intervals B as illustrated in Figure 6(b) that plots the sleep time achieved at typical utilization ($\approx 5\%$) for

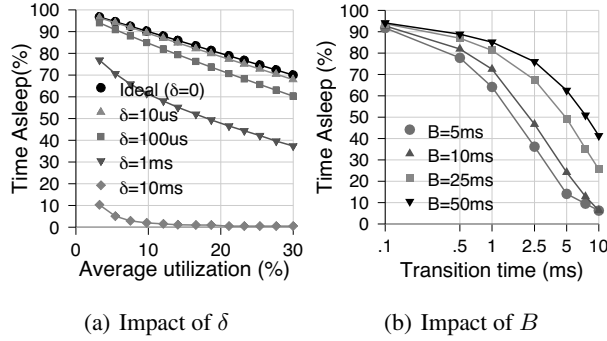


Figure 6: The impact of hardware constants on sleep time.

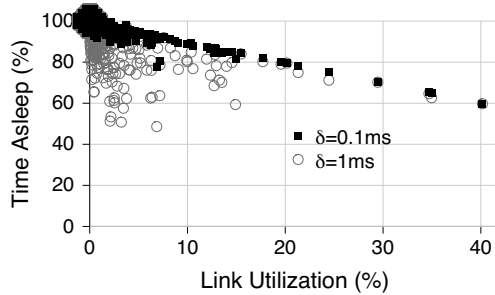


Figure 7: Time asleep per link

increasing transition times and different values of B .

These findings reinforce our intuition that hardware support featuring low-power sleep states and quick transitions (preferably $< 1ms$) between these states are essential to effectively save energy.

Impact of network topology We now evaluate `practB&B` for the Intel enterprise network. The routing structure of the Intel network is strictly hierarchical with a relatively small number of nodes that connect to the wide-area. Because of this we find a wide variation in link utilization – far more than on the Abilene network. Over 77% of links have utilizations below 1% while a small number of links (2.5%) can see significantly higher utilizations of between 20-75%. Correspondingly, the opportunity for sleep also varies greatly across links. This is shown in Figure 7 – each point in the scatter-plot corresponds to a single link and we look at sleep times for two transition times: $0.1ms$ and $1ms$. We see that the dominant trends in sleep time *vs.* utilization remains and that higher δ yields lower savings.

4 Rate-Adaptation in Networks

This section explores the use of performance states to reduce network energy consumption.

4.1 Model and Assumptions

Background In general, operating a device at a lower frequency can enable dramatic reductions in energy consumption for two reasons. First, simply operating more slowly offers some fairly substantial savings.

For example, Ethernet links dissipate between 2-4W when operating between 100Mbps-1Gbps compared to 10-20W between 10Gbps[3]. Second, operating at a lower frequency also allows the use of dynamic voltage scaling (DVS) that reduces the operating voltage. This allows power to scale cubically, and hence energy consumption quadratically, with operating frequency[32]. DVS and frequency scaling are already common in microprocessors for these reasons.

We assume the application of these techniques to network links and associated equipment (i.e., linecards, transceivers). While the use of DVS has been demonstrated in prototype linecards [22], it is not currently supported in commercial equipment and hence we investigate savings under two different scenarios: (1) equipment that supports only frequency scaling and (2) equipment that supports both frequency and voltage scaling.

Model We assume individual links can switch performance states independently and with independent rates for transmission and reception on interfaces. Hence the savings we obtain apply directly to the consumption at the links and interface cards of a network element, although in practice one could also scale the rate of operation of the switch fabric and/or route processor.

We assume that each network interface supports N performance states corresponding to link rates r_1, \dots, r_n (with $r_i < r_{i+1}$ and $r_n = r_{max}$, the default maximum link rate), and we investigate the effect that the granularity and distribution (linear vs. exponential) of these rates has on the potential energy savings.

The final defining characteristic of performance states is the transition time, denoted δ , during which packet transmission is stalled as the link transitions between successive rates. We explore performance for a range of transition times (δ) from 0.1 to 10 milliseconds.

Measuring savings and performance As in the case of sleep we're interested in solutions that reduce the rate at which links operate without significantly affecting performance.

In this section, we use the percentage reduction in average link rate as an indicative measure of energy savings and relate this to overall energy savings in Section 5 where we take into account the power profile of equipment (including whether it supports DVS or not). In terms of performance, we again measure the average and 98th percentile of the end-to-end packet delay and packet loss.

4.2 An optimal strategy

Our initial interest is to understand the extent to which performance states can help if used to best effect. For a DVS processor, it has been shown that the most energy-efficient way to execute C cycles within a given time interval T is to maintain a constant clock speed of

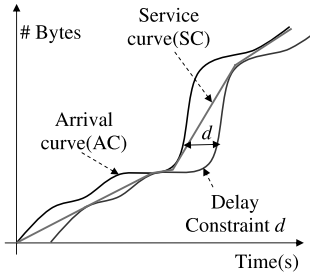


Figure 8: An illustration of delay-constrained service curves and the service curve minimizing energy.

C/T [20]. In the context of a network link, this translates into sending packets at a constant rate equal to the average arrival rate. However under non-uniform traffic this can result in arbitrary delays and hence we instead look for an optimal *schedule* of rates (*i.e.*, the set of rates at which the link should operate at different points in time) that minimize energy while respecting a specified constraint on the additional delay incurred at the link.

More precisely, given a packet arrival curve AC , we look for a service curve SC that minimizes energy consumption, while respecting a given upper bound d on the per-packet queuing delay. The delay parameter d thus serves to tradeoff savings for increased delay. Figure 8(a) shows an example arrival curve and the associated latest-departure curve ($AC+d$), which is simply the arrival curve shifted in time by the delay bound d . To meet the delay constraint, the service curve SC must lie within the area between the arrival and latest-departure curves.

In the context of wireless links, [19] proves that if the energy can be expressed as a convex, monotonically increasing function of the transmission rate, then the minimal energy service curve is the *shortest Euclidean distance* in the arrival space (bytes \times time) between the arrival and shifted arrival curves.

In the scenario where we assume DVS support, the energy consumption is a convex, monotonically increasing function of link rate and thus this result applies to our context as well. Where only frequency scaling is supported, any service curve between the arrival and shifted-arrival curves would achieve the same energy savings and therefore the service curve with the shortest Euclidean distance would be optimal in this case too. In summary, for both frequency-scaling and DVS, the shortest distance service curve would achieve the highest possible energy savings.

Fig. 8 illustrates an example of such a minimal energy service curve. Intuitively, this is the set of lowest constant rates obeying the delay constraint. Note that this per-link optimal strategy is *not* suited to practical implementation since it assumes perfect knowledge of the future arrival curve, link rates of infinite granularity and ignores switching overheads. Nonetheless, it is useful as an estimate of the potential savings by which

to calibrate practical protocols.

We will evaluate the savings achieved by applying the above per-link solution at all links in the network and call this approach `link_optRA`. One issue in doing so is that the service curves at the different links are inter-dependent – *i.e.*, the service curve for a link l depends in turn on the service curves at other links (since the latter in turn determine the arrival curve at l). We address this by applying the per-link optimal algorithm iteratively across all links until the service and arrival curves at the different links converge.

4.3 A practical algorithm

Building on the insight offered by the per-link optimal algorithm, we develop a simple approach, called `practRA` (practical rate adaptation), that seeks to navigate the tradeoff between savings and delay constraints. A practical approach differs from the optimum in that (i) it does not have knowledge of future packet arrivals, (ii) it can only choose among a fixed set of available rates r_1, \dots, r_n , and (iii) at every rate switch, it incurs a penalty δ , during which it cannot send packets.

While knowledge of the future arrival rate is unavailable, we can use the history of packet arrivals to predict the future arrival rate. We denote this predicted arrival rate as \hat{r}_f and estimate it with an exponentially weighted moving average (EWMA) of the measured history of past arrivals. Similarly, we can use the current link buffer size q and rate r_i to estimate the potential queuing delay so as to avoid violating the delay constraint.

With these substitutes, we define a technique inspired by the per-link optimal algorithm. In `practRA`, packets are serviced at a constant rate until we intersect one of the two bounding curves presented earlier (Figure 8): the arrival curve (AC), and the latest-departure curve ($AC+d$). Thus, we avoid increasing the operating rate r_i unless not doing so would violate the delay constraint. This leads to the following condition for rate increases:

A link operating at rate r_i with current queue size q increases its rate to r_{i+1} iff $(\frac{q}{r_i} > d$ OR $\frac{\delta \hat{r}_f + q}{r_{i+1}} > d - \delta$)

The first term checks whether the delay bound d would be violated were we to maintain the current link rate. The second constraint ensures that the service curve does not get too close to the delay-constrained curve which would prevent us from attempting a rate increase in the future without violating the delay bound. That is, we need to allow enough time for a link that increases its rate to subsequently process packets that arrived during the transition time (estimated by $\delta \hat{r}_f$) and its already-accumulated queue. Note that we cannot use delay constraints d smaller than the transition time δ . Similarly, the condition under which we allow a rate decrease is as follows:

A link operating at rate r_i with current queue size q decreases its rate to r_{i-1} iff $q = 0$ AND $\hat{r}_f < r_{i-1}$

First, we only attempt to switch to a lower rate when the queue at the link is empty ($q = 0$). Intuitively, this corresponds to an intersection between the arrival curve and the service curve. However, we don't always switch to a lower rate r_{i-1} when a queue empties as doing so would prevent the algorithm from operating in a (desired) steady state mode with zero queuing delay. Instead, the desirable steady state is one where $r_i > r_f > r_{i+1}$ and we want to avoid oscillating between rates (which would lead to larger average delay). For example, a link that sees a constant arrival rate of 3.5Mbps might oscillate between 3 and 4Mbps (incurring queuing delays at 3Mbps), instead of remaining at 4Mbps (with low average queuing delay). We thus use the additional condition: $\hat{r}_f < r_{i-1}$ to steer our algorithm toward the desired steady state and ensure that switching to a lower rate does not immediately lead to larger queues.

In addition to the above conditions, we further discourage oscillations by enforcing a minimum time $K\delta$ between consecutive switches. Intuitively, this is because rate switching should not occur on timescales smaller than the transition time δ . In our experiments, we found $K = 4$ to be a reasonable value for this parameter.

Note that unlike the `link_optRA` algorithm, the above decision process does not guarantee that the delay constraints will be met since it is based on estimated rather than true arrival rates. Similarly, `practRA` cannot guarantee that the rates used by the links match those used by the link-optimal algorithm. In Section 5, after discussing how power scales with rate, we use simulation to compare our approximate algorithm to the optimal under realistic network conditions. We leave it to future work to analytically bound the inaccuracy due to our approximations.

Finally, we observe that the above rate-adaptation is simple in that it requires no coordination across different nodes, and is amenable to implementation in high-speed equipment. This is because the above decision making need not be performed on a per-packet basis.

4.4 Evaluation

We evaluate the savings vs. performance tradeoff achieved by our practical rate-adaptation algorithm and the impact of equipment and network parameters on the same. We first evaluate the percentage reduction in average link rate achieved by `practRA` for the Abilene network. For comparison, we consider the rate reduction due to `link_optRA` and the upper bound on rate reduction as determined by the average link utilization. In this case, since we found the reduction from `link_optRA` was virtually indistinguishable from the utilization bound, we only show the latter here for clarity. (We re-

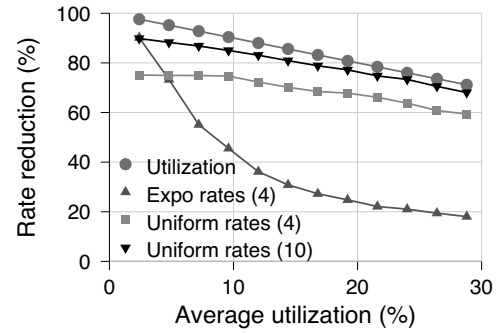


Figure 9: Average rate of operation. The average is weighted by the time spent at a particular rate.

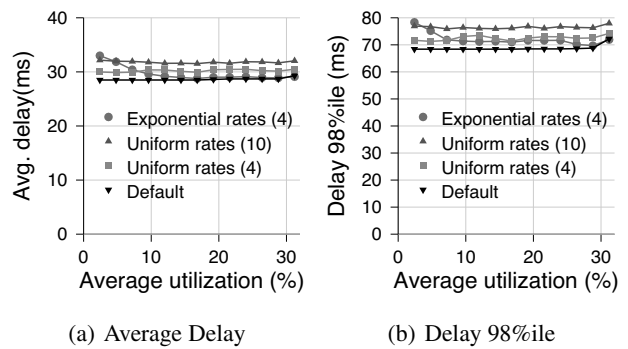
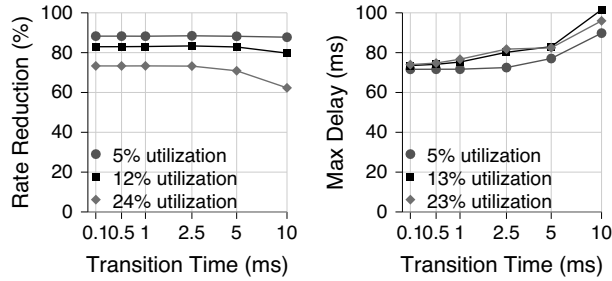


Figure 10: Average and 98th percentile delay achieved by `practRA` for various available rates

port on the energy savings due to `link_optRA` in the following section.) To measure performance, we measure the end-to-end packet delay and loss in a network using `practRA` to that in a network with no rate adaptation.

Impact of hardware characteristics The main constants affecting the performance of `practRA` are (1) the granularity and distribution of available rates, and (2) δ , the time to transition between successive rates. We investigate the reduction in rate for three different distributions of rates r_1, \dots, r_n : (i) 10 rates uniformly distributed between 1Gbps to 10Gbps, (ii) 4 rates uniformly distributed between 1Gbps to 10Gbps and (iii) 4 exponentially distributed rates (10Gbps, 1Gbps, 100Mbps, 10Mbps). We consider the latter case since physical layer technologies for these rates already exist making these likely candidates for early rate-adaptation technologies.

Figure 9 plots the average rate reduction under increasing utilizations, with a per-link delay constraint $d = \delta + 2\text{ms}$, and a transition time $\delta = 1\text{ms}$. We see that for 10 uniformly distributed rates, `practRA` operates links at a rate that approaches the average link utilization. With 4 uniformly distributed rates this reduction drops, but not significantly. However, for exponentially distributed rates, the algorithm performs poorly, indicating that support for uniformly distributed rates is essential.



(a) Rate reduction (time) (b) Max Delay (98th percentile)

Figure 11: Impact of Switch Time (δ) for `practRA`

Figure 10 plots the corresponding average and 98th percentile of the end-to-end delay. We see that, for all the scenarios, the increase in average delay due to `practRA` is very small: ~ 5 ms in the worst case, and less than 2ms for the scenario using 4 uniform rates. The increase in maximum delay is also reasonable: at most 78ms with `practRA`, relative to 69ms with no adaptation.

Perhaps surprisingly, the lowest additional delay occurs in the case of 4 uniformly distributed rates. We found this occurs because there are fewer rate transitions, with corresponding transition delays, in this case. Finally, we found that `practRA` introduced no additional packet loss for the range of utilizations considered.

Next, we look at the impact of transition times δ . Figures 11 (a) and (b) plot the average rate reduction and 98th percentile of delay under increasing δ for different network utilizations. For each test, we set the delay constraint as $d = \delta + 2$ ms, and we assume 10 uniform rates. As would be expected, we see that larger δ lead to reduced savings and higher delay. On the whole we see that, in this scenario, both savings and performance remain attractive for transition times as high as ~ 2 ms.

In summary, these results suggest that rate adaptation as implemented by `practRA` has the potential to offers significant energy savings with little impact on packet loss or delay. In all our tests, we found `practRA` to have minimal effects on the average delay and loss and hence, from here on, we measure the performance impact only in terms of the 98th percentile in packet delay.

Impact of network topology We now evaluate `practRA` applied to the Intel enterprise network. Figure 12 plots the rate reduction across links - each point in the scatter-plot corresponds to a single link, and we look at rate reduction for two rate distribution policies: 10 uniformly distributed rates and 4 exponentially distributed rates. Since these are per-link results, we see significant variations in rate reduction for the same utilization, due to specifics of traffic across various links. We also notice that the dominant trend in reduction remains similar to that seen in the Abilene network (Figure 9).

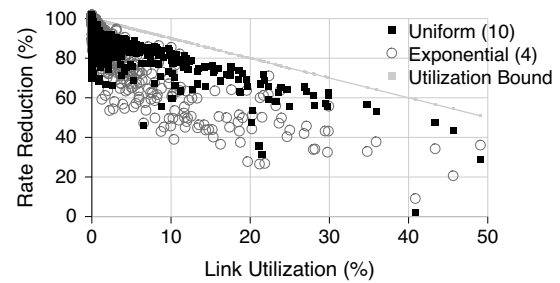


Figure 12: Rate reduction per link

5 Overall Energy Savings

In the previous sections we evaluated power-management solutions based on their ability to increase sleep times (Section 3), or operate at reduced rates (Section 4). In this section, we translate these to *overall energy savings* and hence compare the relative merits of rate adaptation *vs.* sleeping. For this, we develop an analytical model of power consumption under different operating modes. Our model derives from measurements of existing networking equipment [13, 5, 24]. At the same time, we construct the model to be sufficiently general that we may study the potential impact of future, more energy-efficient, hardware.

5.1 Power Model

Recall from section 2 that the total energy consumption of a network element operating in the absence of any power-saving modes can be approximated as: $E = p_a T_a + p_i T_i$.

We start by considering the power consumption when actively processing packets (p_a). Typically, a portion of this power draw is static in the sense that it does not depend on the operating frequency (e.g., refresh power in memory blocks, leakage currents and so forth) while the dominant portion of power draw does scale with operating frequency. Correspondingly, we set:

$$p_a(r) = C + f(r) \quad (2)$$

Intuitively, C can be viewed as that portion of power draw that cannot be eliminated through rate adaptation while $f(r)$ reflects the rate-dependent portion of energy consumption. To reflect the relative proportions of C and $f(r)$ we set C to be relatively small – between 0.1 and 0.3 of the maximum active power $p_a(r_n)$. To study the effect of just frequency scaling alone we set $f(r) = O(r)$ and set $f(r) = O(r^3)$ to evaluate dynamic voltage scaling (DVS). In evaluating DVS, we need to consider an additional constraint – namely that, in practice, there is a minimum rate threshold below which scaling the link rate offers no further reduction in voltage. We thus define a maximum scaling factor λ and limit our choice of available operating rates to lie between $[r_n/\lambda, r_n]$, for scenar-

ios that assume voltage scaling. While current transistor technology allows scaling up to factors as high as 5 [32], current processors typically use $\lambda \sim 2$ and hence we investigate both values as potential rate scaling limits.

Empirical measurements further reveal that the idle mode power draw, p_i , varies with operating frequency in a manner similar to the active-mode power draw but with lower absolute value [13]. Correspondingly, we model the idle-mode power draw as:

$$p_i(r) = C + \beta f(r) \quad (3)$$

Intuitively, the parameter β represents the relative magnitudes of routine work incurred even in the absence of packets to the work incurred when actively processing packets. While measurements from existing equipment suggest values of β as high as 0.8 for network interface cards [13] and router linecards [5], we would like to capture the potential for future energy-efficient equipment and hence consider a wide range of β between [0.1, 0.9].

Energy savings from rate adaptation With the above definitions of p_a and p_i , we can now evaluate the overall energy savings due to rate adaptation. The total energy consumption is now given by:

$$\sum_{r_k} p_a(r_k) T_a(r_k) + p_i(r_k) T_i(r_k) \quad (4)$$

Our evaluation in Section 4 yields the values of $T_a(r_k)$ and $T_i(r_k)$ for different r_k and test scenarios, while Eqns. 2 and 3 allow us to model $p_a(r_k)$ and $p_i(r_k)$ for different C , β and $f(r)$. We first evaluate the energy savings using rate adaptation under frequency scaling ($f(r) = O(r)$) and DVS ($f(r) = O(r^3)$). For these tests, we set C and β to middle-of-the-range values of 0.2 and 0.5 respectively; we examine the effect of varying C and β in the next section.

Figure 13(a) plots the energy savings for our practical (`practRA`) and optimal (`link_optRA`) rate adaptation algorithms assuming only frequency scaling. We see that, in this case, the relative energy savings for the different algorithms as well as the impact of the different rate distributions is similar to our previous results (Fig. 9) that measured savings in terms of the average reduction in link rates. Overall, we see that significant savings are possible even in the case of frequency scaling alone.

Figure 13(b) repeats the above test assuming voltage scaling for two different values of λ , the maximum rate scaling factor allowed by DVS. In this case, we see that the use of DVS significantly changes the savings curve – the more aggressive voltage scaling allows for larger savings that can be maintained over a wide range of utilizations. Moreover, we see that once again the savings from our practical algorithm (`practRA`) approach those from the optimal algorithm. Finally, as

expected, increasing the range of supported rates (λ) results in additional energy savings.

Energy savings from sleeping To model the energy savings with sleeping, we need to pin down the relative magnitudes of the sleep mode power draw (p_s) relative to that when idle (p_i). We do so by introducing a parameter γ and set:

$$p_s = \gamma p_i(r_n) \quad (5)$$

where $0.0 \leq \gamma \leq 1.0$. While the value of γ will depend on the hardware characteristics of the network element in question, empirical data suggest that sleep mode power is typically a very small fraction of the idle-mode power consumption: ~ 0.02 for network interfaces [13], 0.001 for RFM radios [11], 0.3 for PC cards [11] and less than 0.1 for DRAM memory [8]. In our evaluation we consider values of γ between 0 and 0.3.

With this, the energy consumption of an element that spends time T_s in sleep is given by:

$$E = p_a(r_n) T_a + p_i(r_n) (T_i - T_s) + p_s T_s. \quad (6)$$

Our evaluation from Section 3 estimated T_s for different scenarios. Figure 13(c) plots the corresponding overall energy savings for different values of γ for our `practB&B` algorithm. We assume a transition time $\delta = 1\text{ms}$, and a buffering interval $B = 10\text{ms}$. Again, our results confirm that sleeping offers good overall energy savings and that, as expected, energy savings are directly proportional to γ .

5.2 Comparison: Sleep vs. Rate Adaptation

We now compare the savings from sleeping vs. rate adaptation by varying the two defining axes of our power model: C , the percentage of power that does not scale with frequency, and β that determines the relative magnitudes of idle to active power draws. We consider two end-of-the-range values for each: $C = 0.1$ and $C = 0.3$ and $\beta = 0.1$ and $\beta = 0.8$. Combining the two gives us four test cases that span the spectrum of hardware power profiles:

- $C = 0.1$ and $\beta = 0.1$: captures the case where the static portion of power consumption (that cannot be rate-scaled away) is low and idle-mode power is significantly lower than active-mode power.
- $C = 0.1$ and $\beta = 0.8$: the static portion of power consumption is low and idle-mode power is almost comparable to active-mode power.
- $C = 0.3$ and $\beta = 0.1$: the static portion of power consumption is high; idle-mode power is significantly lower than that in active mode.
- $C = 0.3$ and $\beta = 0.8$: the static portion of power consumption is high; idle-mode power is almost comparable to active-mode power.

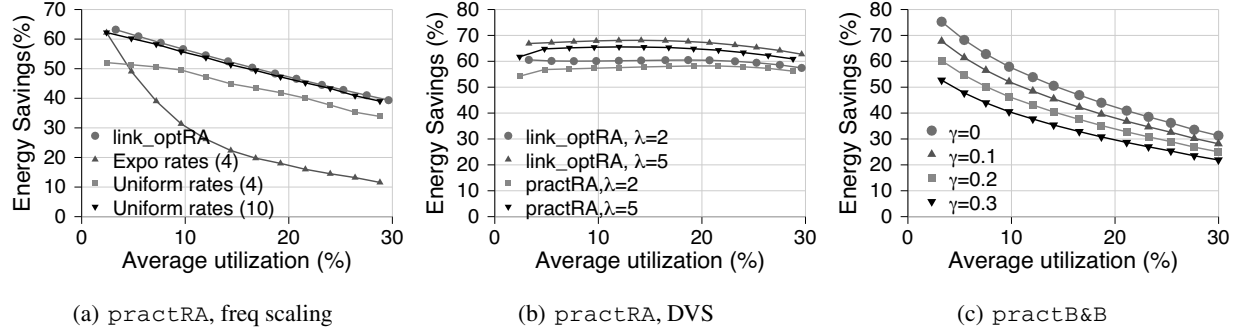


Figure 13: Total Energy Saving with Sleeping and Rate Adaptation

We evaluate energy savings for each of the above scenarios for the case where the hardware supports DVS and when the hardware only supports frequency scaling.

With DVS: $f(r) = O(r^3)$ Figures 14 plots the overall energy savings for practRA and practB&B for the different test scenarios. These tests assume 10 uniformly distributed rates and a sleep power $p_s = 0.1p_t(r_n)$. In each case, for both sleep and rate-adaptation, we consider hardware parameters that reflect the best and worst case savings for the algorithm in question. For practRA, these parameters are λ (the range for voltage scaling) and δ (the transition time). For the best-case results these are $\lambda = 5$ and $\delta = 0.1ms$; for the worst case: $\lambda = 2$, $\delta = 1ms$. The parameter for practB&B is the transition time δ which we set as $\delta = 0.1ms$ (best case) and $\delta = 1ms$ (worst case).

The conclusion we draw from Figure 14 is that, in each scenario there is a “boundary” utilization below which sleeping offers greater savings, and above which rate adaptation is preferable. Comparing across graphs, we see that the boundary utilization depends primarily on the values of C and β , and only secondarily on the transition time and other hardware parameters of the algorithm. For example, the boundary utilization for $C = 0.1$ and $\beta = 0.1$ varies between approximately 5-11% while at $C = 0.3$, $\beta = 0.8$ this boundary utilization lies between 4% and 27%. We also evaluated savings under different traffic characteristics (CBR, Pareto) and found that the burstiness of traffic has a more secondary effect on the boundary utilization.

For further insight on what determines the boundary utilization, we consider the scenario of a single idealized link. The sleep-mode energy consumption of such an idealized link can be viewed as:

$$E_{sleep} = p_a(r_{max})\mu T + p_s(1 - \mu)T \quad (7)$$

Similarly, the idealized link with rate adaptation is one that runs with an average rate of μr_{max} for an energy consumption of:

$$E_{rate} = p_a(\mu r_{max})T \quad (8)$$

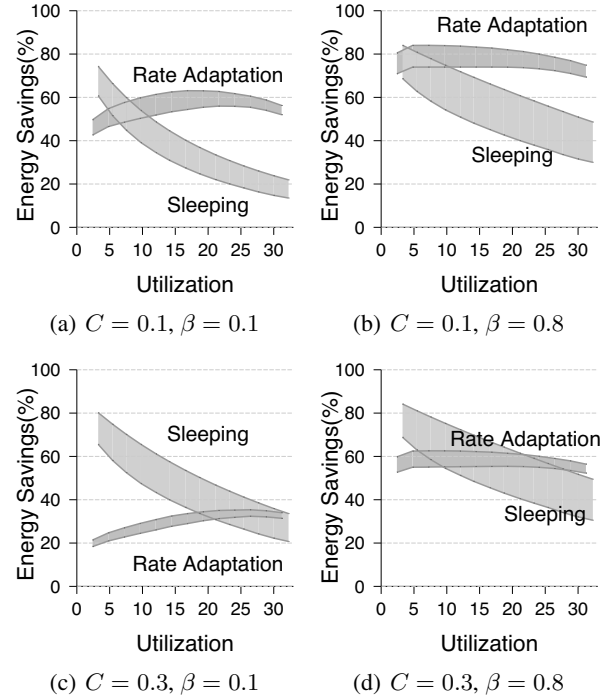


Figure 14: Comparison of energy savings between sleep and rate adaptation. Support for dynamic voltage scaling.

Figure 15 represents the boundary utilization for this idealized link as a function of C . In this idealized scenario, the dominant parameter is C because the link is never idle and therefore β has only a small, indirect effect on p_s . The gray zone in the figure represents the spread in boundary utilization obtained by varying β between 0.1 and 0.9.

With frequency scaling alone: $f(r) = O(r)$ Figures 16 plots the overall energy savings for practRA and practB&B for the different test scenarios in the more pessimistic scenario where voltage scaling is not supported. Due to lack of space, we only plot the comparison for the first two test scenarios where $C = 0.1$; at $C = 0.3$, the savings show a similar scaling trend but with significantly poorer performance for rate-adaptation

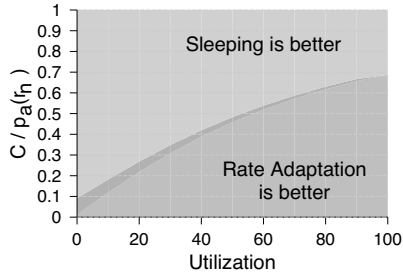


Figure 15: Sleeping vs. rate-adaptation

and hence add little additional information.

The primary observation is that the savings from rate adaptation are significantly lower than in the previous case with DVS and, in this case, sleeping outperforms rate adaptation more frequently. We also see that – unlike the DVS case – network utilization impacts energy savings in a similar manner for both sleeping and rate-adaptation (*i.e.*, the overall “slope” of the savings-vs-utilization curves is similar with both sleeping and rate-adaptation while they were dramatically different with DVS – see Fig. 14).

Once again, we obtain insight on this by studying the the highly simplified case of a single idealized link. For this idealized scenario with $f(r) = O(r)$, we find that the boundary condition that determines whether to use sleep or rate adaptation is in fact independent of network utilization. Instead, one can show that sleep is superior to rate-adaptation if the following inequality holds:

$$c > \frac{\gamma\beta}{1 - \gamma(1 - \beta)} \quad (9)$$

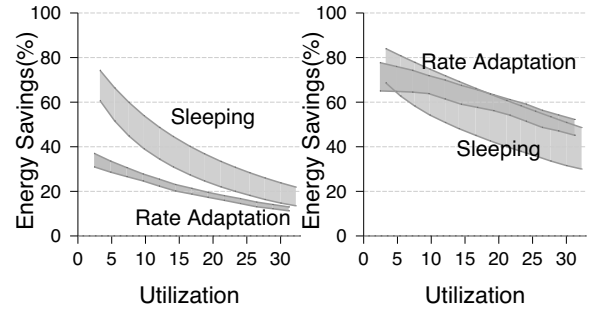
Otherwise, rate adaptation is superior.

In practice, network utilization does play a role (as our results clearly indicate) because the various practical constraints due to delay bounds and transition times prevent our algorithms from fully exploiting all opportunities to sleep or change rates.

In summary, we find that both sleeping and rate-adaptation are useful, with the tradeoff between them depending primarily on the power profile of hardware capabilities and network utilization. Results such as those presented here can guide operators in deciding how to best run their networks. For example, an operator might choose to run the network with rate adaptation during the day and sleeping at night based on where the boundary utilization intersects diurnal behavior, or identify components of the network with consistently low (or high) utilization to be run with sleeping (or rate-adaptation).

6 Related Work

There is a large body of work on power management in contexts complementary to ours. This includes power provisioning and load balancing in data centers[6, 9],



(a) $C = 0.1, \beta = 0.1$

(b) $C = 0.1, \beta = 0.8$

Figure 16: Energy savings of sleep vs. rate adaptation, $\beta = 0.1$, frequency scaling alone.

and OS techniques to extend battery lifetimes in mobiles [10, 30].

Perhaps the first to draw attention to the problem of saving overall energy in the network was an early position paper by Gupta *et al.* [12]. They use data from the US Department of Commerce to detail the growth in network energy consumption and argue the case for energy-saving network protocols, including the possibility of wake-on-arrival in wired routers. In follow-on work they evaluate the application of opportunistic sleeping in a campus LAN environment [21, 11].

Other recent work looks at powering-down redundant access points (APs) in enterprise wireless networks [17]. The authors propose that a central server collect AP connectivity and utilization information to determine which APs can be safely powered down. This approach is less applicable to wired networks that exhibit much less redundancy.

Sleeping has also been explored in the context of 802.11 to save client power, *e.g.*, see [2]. The 802.11 standard itself includes two schemes (Power-Save Poll and Automatic Power Save Delivery) by which access points may buffer packets so that clients may sleep for short intervals. In some sense, our proposal for bunching traffic to improve sleep opportunities can be viewed as extending this idea deep into the network.

Finally, the IEEE Energy Efficient Ethernet Task Force has recently started to explore both sleeping and rate adaptation for energy savings. Some initial studies consider individual links and are based on synthetic traffic and infinite buffers [4].

In the domain of sensor networks, there have been numerous efforts to design energy efficient protocols. Approaches investigated include putting nodes to sleep using TDMA-like techniques to coordinate transmission and idle times (*e.g.*, FPS [14]), and distributed algorithms for sleeping (*e.g.*, S-MAC [28]). This context differs from ours in many ways.

7 Conclusion

We have argued that power management states that slow down links and put components to sleep stand to save much of the present energy expenditure of networks. At a high-level, this is apparent from the facts that while network energy consumption is growing networks continue to operate at low average utilizations. We present the design and evaluation of simple power management algorithms that exploit these states for energy conservation and show that – with the right hardware support – there is the potential for saving much energy with a small and bounded impact on performance, *e.g.*, a few milliseconds of delay. We hope these preliminary results will encourage the development of hardware support for power saving as well as algorithms that use them more effectively to realize greater savings.

Aknowledgments

We thank Robert Hays, Bob Grow, Bruce Nordman, Rabin Patra and Ioan Bejenaru for their suggestions. We also thank the anonymous reviewers and our shepherd Jon Crowcroft for their useful feedback.

References

- [1] Power and Thermal Management in the Intel Core Duo Processor. In *Intel Technology Review, Volume 10, Issue 2, Section 4*. 2006.
- [2] Y. Agarwal, R. Chandra, et al. Wireless Wakeups Revisited: Energy Management for VoIP over Wi-Fi Smartphones. In *ACM MobiSys*. 2007.
- [3] C. Gunaratne, K. Christensen and B. Nordman. Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed. In *International Journal of Network Management*. October 2005.
- [4] C. Gunaratne, K. Christensen et al. Reducing the Energy Consumption of Ethernet with Adaptive Link Rate (ALR). In *IEEE Transactions on Computers*. April 2008.
- [5] J. Chabarek, J. Sommers, et al. Power Awareness in Network Design and Routing. In *IEEE INFOCOM*. 2008.
- [6] J. S. Chase, D. C. Anderson, et al. Managing Energy and Server Resources in Hosting Centers. In *ACM SOSP*. 2001.
- [7] Cisco Systems. NetFlow Services and Applications. White Paper, 2000.
- [8] X. Fan, C. S. Ellis, et al. Memory Controller Policies for DRAM Power Management. In *International Symposium on Low Power Electronics and Design*. 2003.
- [9] X. Fan, W.-D. Weber, et al. Power Provisioning for a Warehouse-Sized Computer. In *ACM ISCA*. 2007.
- [10] J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *ACM SOSP*. 1999.
- [11] M. Gupta, S. Grover, et al. A Feasibility Study for Power Management in LAN Switches. In *ICNP*. 2004.
- [12] M. Gupta and S. Singh. Greening of the Internet. In *ACM SIGCOMM, Karlsruhe, Germany*. August 2003.
- [13] R. Hays. Active/Idle Toggling with Low-Power Idle, http://www.ieee802.org/3/az/public/jan08/hays_01_0108.pdf. In *IEEE 802.3az Task Force Group Meeting*. 2008.
- [14] B. Hohlt, L. Doherty, et al. Flexible Power Scheduling for Sensor Networks. In *IEEE and ACM Third International Symposium on Information Processing in Sensor Networks (IPSN)*. April 2004.
- [15] IEEE 802.3 Energy Efficient Ethernet Study Group. http://grouper.ieee.org/groups/802/3/eee_study/.
- [16] Ipmn Sprint. The Applied Research Group. <http://ipmon.sprint.com/>.
- [17] A. Jardosh et al. Towards an Energy-Star WLAN Infrastructure. In *HOTMOBILE*. 2007.
- [18] S. Kandula, D. Katabi, et al. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *ACM SIGCOMM 2005*.
- [19] M. Lin and Y. Ganjali. Power-Efficient Rate Scheduling in Wireless Links Using Computational Geometric Algorithms. In *IWCMC*. 2006.
- [20] J. Lorch. Operating Systems Techniques for Reducing Processor Energy Consumption. In *Ph.D. Thesis, University of California, Berkeley*. 1994.
- [21] M. Gupta and S. Singh. Dynamic Ethernet Link Shutdown for Energy Conservation on Ethernet Links. In *IEEE ICC*. 2007.
- [22] M. Mandviwalla and N.-F. Tzeng. Energy-Efficient Scheme for Multiprocessor-Based Router Linecards. In *IEEE SAINT*. 2006.
- [23] E. Miranda and L. McGarry. Power/Thermal Impact of Networking Computing. In *Cisco System Research Symposium, August, 2006*.
- [24] S. Nedeveschi. Reducing Network Energy Consumption via Sleeping and Rate-adaptation, http://www.ieee802.org/3/az/public/jan08/nedeveschi_01_0108.pdf. In *IEEE 802.3az Task Force Group Meeting*. 2008.
- [25] B. Nordman. Energy Efficient Ethernet, Outstanding Questions. 2007.
- [26] K. W. Roth, F. Goldstein, et al. Energy Consumption by Office and Telecommunications Equipment in Commercial Buildings - Volume I: Energy Consumption Baseline. Tech. Rep. 72895-00, Arthur D. Little, Inc, Jan. 2002.
- [27] The Abilene Observatory. <http://abilene.internet2.edu/observatory>.
- [28] W. Ye, J. Heidemann, et al. An Energy-efficient MAC Protocol for Wireless Sensor Networks. In *IEEE INFOCOM*. 2002.
- [29] Yin Zhang's AbileneTM, <http://www.cs.utexas.edu/~yzhang/research/AbileneTM>.
- [30] W. Yuan and K. Nahrstedt. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In *ACM SOSP*. 2003.
- [31] M. Yuksel, B. Sikdar, et al. Workload generation for ns Simulations of Wide Area Networks and the Internet. In *Communication Networks and Distributed Systems Modeling and Simulation Conference*. 2000.
- [32] B. Zhai, D. Blaauw, et al. Theoretical and Practical Limits of Dynamic Voltage Scaling. In *DAC*. 2004.

Notes

¹In reality the energy savings using rate-adaptation will depend on the *distribution* of operating rates over time and the corresponding power consumption at each rate. For simplicity, we initially use the average rate of operation as an indirect measure of savings in Section 4 and then consider the complete distribution of operating rates in Section 5 when we compute energy savings.

Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services

Gong Chen^{*}, Wenbo He^{*}, Jie Liu[†], Suman Nath[†], Leonidas Rigas[‡], Lin Xiao[†], Feng Zhao[†]

^{*}*Dept. of Statistics, University of California, Los Angeles, CA 90095*

^{*}*Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL 61801*

[‡]*Microsoft Research, One Microsoft Way, Redmond, WA 98052*

gchen@stat.ucla.edu, webohe@uiuc.edu

{liuj, sumann, leonr, lixiao, zhao}@microsoft.com

Abstract

Energy consumption in hosting Internet services is becoming a pressing issue as these services scale up. Dynamic server provisioning techniques are effective in turning off unnecessary servers to save energy. Such techniques, mostly studied for request-response services, face challenges in the context of connection servers that host a large number of long-lived TCP connections. In this paper, we characterize unique properties, performance, and power models of connection servers, based on a real data trace collected from the deployed Windows Live Messenger. Using the models, we design server provisioning and load dispatching algorithms and study subtle interactions between them. We show that our algorithms can save a significant amount of energy without sacrificing user experiences.

1 Introduction

Internet services such as search, web-mail, online chatting, and online gaming, have become part of people's everyday life. Such services are expected to scale well, to guarantee performance (e.g., small latency), and to be highly available. To achieve these goals, these services are typically deployed in clusters of massive number of servers hosted in dedicated data centers. Each data center houses a large number of heterogeneous components for computing, storage, and networking, together with an infrastructure to distribute power and provide cooling.

Viewed from the outside, a data center is a "black box" that responds to a stream of requests from the Internet, while consuming power from the electrical grid and producing waste heat. As the demand on Internet services drastically increases in recent years, the energy used by data centers, directly related to the number of hosted servers and their workload, has been skyrocketing [8]. In 2006, U.S. data centers consumed an estimated 61 billion kilowatt-hours (kWh) of energy, enough to power 5.8 million average US households.

Data center energy savings can come from a number of places: on the hardware and facility side, e.g., by designing energy-efficient servers and data center infrastructures, and on the software side, e.g., through resource management. In this paper, we take a software-based approach, consisting of two interdependent techniques: *dynamic provisioning* that dynamically turns on a minimum number of servers required to satisfy application-specific quality of service, and *load dispatching* that distributes current load among the running machines. Our approach is motivated by two observations from real data sets collected from operating Internet services. First, the total load of a typical Internet service fluctuates over a day. For example, the fluctuation for the number of users logged on to Windows Live Messenger can be about 40% of the peak load within a day. Similar patterns were found in other studies [3, 20]. Second, an active server, even when it is kept idle, consumes a non-trivial amount of power (> 66% of the peak from our measurements, similar to other studies [9]). The first observation provides us the opportunity to dynamically change the number of active servers, while the second observation implies that shutting down machines during off-peak period provides maximum power savings.

Both dynamic provisioning and load dispatching have been extensively studied in literature [20, 3, 6]. However, prior work studies them separately since the main foci were on *request-response* type of services, such as Web serving. Simple Web server transactions are typically short. There is no state to carry over when the total number of servers changes. In that context, first the minimum number of servers sufficient to handle the current request rate is determined. After the required number of servers are turned on, incoming requests are typically distributed evenly among them [3].

Many Internet services keep long-lived connections. For example, HTTP1.1 compatible web servers can optionally keep the TCP connection after returning the initial web request in order to improve future response per-

formance. An extreme case is connection-oriented services like in instant messaging, video sharing, Internet games, and virtual life applications, where users may be continuously logged in for hours or even days. For a connection server, the number of connections on a server is an integral of its net login rate (gross login rate minus logout rate) over the time it has been on. The power consumption of the server is a function of many factors such as number of active connections and login rate. Unlike request-response servers that serve short-lived transactions, long-lived connections in connection servers present unique characteristics:

1. The capacity of a connection server is usually constrained by both the rate at which it can accept new connections and the total number of active connections on the server. Moreover, the maximum tolerable connection rate is typically much smaller than the total number of active connections due to several reasons such as expensive connection setup procedure and conservative SLA (Service Level Agreement) with back-end authentication services or user profiles. This implies that when a new server is turned on, it cannot be fully utilized immediately like simple web servers. The load, in terms of how many users it hosts, can only increase gradually.
2. If the number of server is under provisioned, new login requests will be rejected and users receive “service not available” (SNA) errors. When a connection server with active users is turned off, users may experience a short period of disconnection, called “server initiated disconnections” (SID). When this happens, the client software typically tries to reconnect back, which may create an artificial surge on the number of new connections, and generate unnecessary SNAs. Both errors should be avoided to preserve user experiences.
3. As we will see in Section 3.5, the energy cost of maintaining a connection is orders of magnitude smaller than processing a new connection request. Thus, a provisioning algorithm that turns off a server with a large number of connections, which in turn creates a large number of reconnections, may defeat the purpose of energy saving.

Due to these unique properties, existing dynamic provisioning and load dispatching algorithms designed for request-response services may not work well with connection services. For example, consider a simple strategy that dynamically turns on or off servers based on current number of users and then balances load among all active servers [3]. Since a connection server takes time, say T , to be fully utilized, provisioning X new servers based on *current users* may not be sufficient. Note that

newly booted servers will require T time to take the target load, and during this time the service will have fewer fully utilized servers than needed, causing poor service. Moreover, after time T of turning on X new servers, workload can significantly change, making these X new servers either insufficient or unnecessary. Consider, on the other hand, that a server with N connections is turned off abruptly when the average connected users per server is low. All those users will be disconnected. Since many disconnected clients will automatically re-login, currently available servers may not be able to handle the surge. In short, a reactive provisioning system is very likely to cause poor service or create instability. This can be avoided by a *proactive* algorithm that takes the transient behavior into account. For example, we can employ a load prediction algorithm to turn on machines gradually before they are needed and to avoid turning on unnecessary machines to cope with temporary spikes in load. At the same time, the provisioning algorithm needs to work together with load dispatching mechanism and anticipate the side effect of changing server numbers. For example, if loads on servers are intentionally skewed, instead of balanced, to create “tail” servers with fewer live connections, then turning off a tail server will not generate any big surge to affect login patterns.

In this paper, we develop power saving techniques for connection services, and evaluate the techniques using data traces from Windows Live Messenger (formerly MSN Messenger), a popular instant messaging service with millions of users. We consider server provisioning and load dispatching in a single framework, and evaluate various load skewing techniques to trade off between energy saving and quality of service. Although the problem is motivated by Messenger services, the results should apply to other connection-oriented services. The **contributions** of the paper are:

- We characterize performance, power, and user experience models for Windows Live Messenger connection servers based on real data collected over a period of 45 days.
- We design a common provisioning framework that trades off power saving and user experiences. It takes into account the server transient behavior and accommodates various load dispatching algorithms.
- We design load skewing algorithms that allow significant amount of energy saving (up to 30%) without sacrificing user experiences, i.e., maintaining very small number of SIDs.

The rest of paper is organized as follows. Section 2 discusses related work. In Section 3, we give a brief overview of the Messenger connection server behavior and characterize connection server power, performance,

and user experience models. We present the load prediction and server provisioning algorithms in Section 4, and load dispatching algorithms in Section 5. Using data traces from deployed Internet services, we evaluate various algorithms and show the results in Section 6. Finally, we conclude the paper with discussions in Section 7.

2 Related Work

In modern CPUs, P-states and clock modulation mechanisms are available to control CPU power consumption according to the load at any particular moment. Dynamic Voltage/Frequency Scaling (DVFS) has been developed as a standard technique to achieve power efficiency of processors [19, 13, 17]. DVFS is a powerful adaptation mechanism, which adjusts power provisioning according to workload in computing systems. A control-based DVFS policy combined with request batching has been proposed in [7], which trades off system responsiveness to power saving and adopts a feedback control framework to maintain a specified response time level. A DVFS policy is implemented in [21] on a stand-alone Apache web server, which manages tasks to meet soft real-time deadlines. Flautner et al. [10] adopts performance-setting algorithms for different workload characteristics transparently, and implements the DVFS policy on per-task basis.

A lot of efforts have been made to address power efficiency in data centers. In [11], Ganesh, et al. proposed a file system based solution to reduce disk array energy consumption. The connection servers we study in this paper have little disk IO load. Our focus is on provisioning entire servers. In [20], Pinheiro et al. presented a simple policy to turn cluster nodes on and off dynamically. Chase et al. [3] allocated computing resources based on an economic approach, where services “bid” for resources as a function of required performance, and the system continuously monitors load and allocates resources based on its utility. Heath et al. [14] studied Web service on-off strategies in the context of heterogeneous server types, although focusing on only short transactions. Various other work using adaptive policies to achieve power efficiency have been proposed. Abdelzaher et al. in [1] employed a Proportional-Integration (PI) controller for an Apache Web server to control the assigned processes (or threads) and to meet soft realtime latency requirements. Other work based on feedback and/or optimal control theory includes [6, 5, 23, 22, 16, 15], which attempt to dynamically optimize for energy, resources and operational costs while meeting performance-based SLAs.

In contrast to previous work, we consider and design load dispatching and dynamic provisioning schemes together since we observe that, in the context of con-

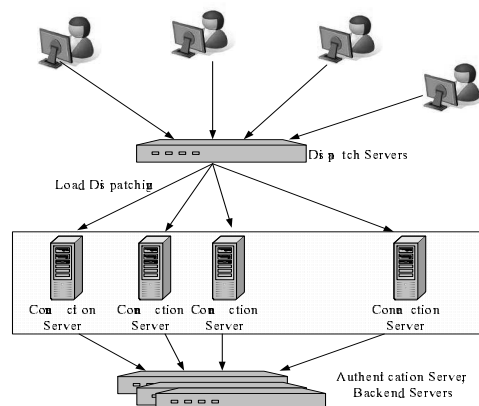


Figure 1: Connection service architecture.

nection servers, they have subtle interaction with each other. These two components work together to control power consumption and performance levels. Moreover, we adopt forecast-based and hysteresis-based provisioning to bear with slow rebooting of servers.

3 Connection Servers Background

Connection servers are essential for many Internet services. In this paper, we consider dedicated connection servers, each of which runs only one connection service application. However, the discussion can be generalized to servers hosting multiple services as well.

3.1 Connection Service

Figure 1 shows an example of the front door architecture for connection intensive Internet applications. Users, through dedicated clients, applets, or browser plug-ins, issue login requests to the service cloud. These login requests first reach a dispatch server (DS), which picks a connection server (CS) and returns its IP address to the client. The client then directly connects to the CS. The CS authenticates the user and if succeeded, a live TCP connection is maintained between the client and the CS until the client logs off. The TCP connection is usually used to update user status (e.g. on-line, busy, off-line, etc.) and to redirect further activities such as chatting and multimedia conferencing to other back-end servers.

At the application level, each CS is subject to two major constraints: the maximum login rate and the maximum number of sockets it can host. The new user login rate L is defined as the number of new connection requests that a CS processes in a second. A limit on login rate L_{\max} is set to protect CS and other back-end services. For example, Messenger uses Windows Live ID (a.k.a. Passport) service for user authentication. Since Live ID is also shared by other Microsoft and non-Microsoft applications, a service-level agreement (SLA)

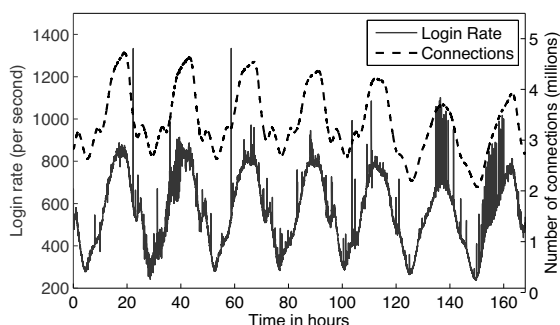


Figure 2: One week of load pattern (Monday to Sunday) from Windows Live Messenger connection servers

is set to bound the number of authentication requests that Messenger can forward. In addition, a new user login is a CPU intensive operation on the CS, as we will show in Section 3.5. Having L_{max} protects the CS from being overloaded or entering unsafe operation regions.

In terms of concurrent live TCP connections, a limit N_{max} is set on the total number of sockets for each CS for two reasons: the memory constraints and the fault tolerance concerns. Maintaining a TCP socket is relatively cheap for the CPU but requires a certain amount of memory. At the same time, if a CS crashes, all its users will be disconnected. As most users set their clients to automatically reconnect when disconnected, a large number of new login requests will hit the servers. Since there is a limit on new login rate, not all reconnect requests can be processed in a short period of time, creating undesirable user experiences.

3.2 Load Patterns

Popular connection servers exhibit periodic load patterns with large fluctuation, similar to those reported in [6].

Figure 2 shows a pattern of the login rates and total number of connections to the Messenger service over the course of a week. Only a subset of the data, scaled to 5 million connections on 60 connection servers, is reported here. It is clear that the number of connections fluctuates over day and night times. In fact, for most days, the amount of fluctuation is about 40% of the corresponding peak load. Some geo-distributed services show even larger fluctuation. The login rates for the same time period are scaled similarly. It is worth noting that the login rate is noisier than the connection count.

Since the total number of servers must be provisioned to handle peak load to guarantee service availability, it is a “overkill” when the load is low. If we can adapt server resource utilization accordingly, we should be able to save substantial energy. In addition, the smooth pattern indicates that the total number of connections is predictable. In fact, if we look at the pattern over weeks, the

similarity is more significant. Since turning on servers takes time, the prediction will help us act early.

3.3 Power Model

Understanding how power are consumed by connection servers provides us insights on energy saving strategies. Connection servers are CPU, network, and memory intensive servers. There is almost no disk IO in normal operation, except occasional log writing. Since memory is typically pre-allocated to prevent run-time performance hit, the main contributor to the power consumption variations of a server is the CPU utilization.

We measured power consumption of typical servers while changing the CPU utilization by using variable workloads. Figure 3 shows the power consumption on two types of servers, where the horizontal axis indicates the average CPU utilization reported by the OS, and the vertical axis indicates the average power consumption measured at the server power plug.

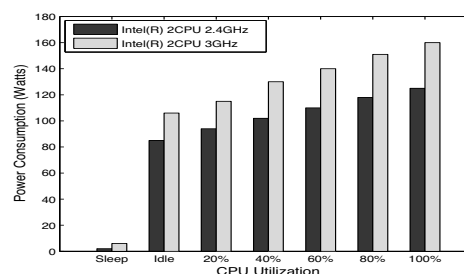


Figure 3: Power consumption v.s. CPU utilization

We observe two important facts. First, the power consumption increases almost linearly with CPU utilization, as reported in other studies [9]. Second, an idle server consumes up to 66% of the peak power, because even when a server is not loaded with user tasks, the power needed to run the OS and to maintain hardware peripherals, such as memory, disks, master board, PCI slots, and fans, is not negligible. Figure 3 implies that if we pack connections and login requests to a portion of servers, and keep the rest of servers hibernating (or shutting-down), we can achieve significant power savings. However, the consolidation of login requests results in high utilization of those servers, which may downgrade performance and user experiences. Hence, it is important to understand the user experience model before we address power saving schemes for large-scale Internet service.

3.4 User Experiences

In the context of connection servers, we consider the following factors as user experience metric.

1. Service Not Available (SNA) Error. Service unavailable is an error message returned to the client software when there is not enough resource to handle user login.

This happens when DS cannot find a CS that can take the new login request, or when a CS receives a login request, it observes that its login rate or number of existing sockets exceeds corresponding limits.

2. Server-Initiated Disconnection (SID). A disconnection happens when a live user socket is terminated before the user issues a log off request. This can be caused by network failures between the client and the server, by server crash, or by the server sending a “reconnect” request to the client. Since network failure and server crash are not controlled by the server software, we use server-initiated disconnection (SID) as the metric for short term user experience degradation. SID is usually a part of connection server protocol, like MSNP [18], so that a server can be shut down gracefully (for software update for example). SID can be handled transparently by the client software so that it is un-noticeable to users. Nevertheless, some users, if they disable automated reconnect or are in active communication with their buddies, may observe transient disconnections.

3. Transaction Latency. Research on quality of services, e.g. [21, 6], often uses transaction delays as a metric for user experiences. However, after we examined connection server transactions, including user login, authentication, messaging redirection, etc., we observe no direct correlation between transaction delays and server load (in terms of CPU, number of connection, and login rate). The reason is that connection services are not CPU bounded. The load of processing transactions is well under control when the number of connection and the login rates are bounded. So, we will not consider transaction delays as a quality of service metric in this paper.

From this architectural description, it is clear that the DS and its load dispatching algorithm play a critical role in the shape of the load in connection servers. This motivates us to focus on the interaction between CS and DS, the provisioning algorithms and load dispatching algorithms to achieve power saving while maintaining user experiences in terms of SNA and SID.

3.5 Performance Model

To characterize the effect of load dispatching on service loads, it is important to understand the relationship between application level parameters such as user login and physical parameters such as CPU utilization and power consumption. In other words, we need to identify the variables that significantly affect CPU and power. This would enable us to control CPU usage and power consumption of the servers by controlling these variables.

We use a workload trace of Messenger service to identify the key variables affecting CPU usage and power. The trace contains 32 different performance counters such as login rate, connection count, memory usage, CPU usage, connection failures, etc. of all production

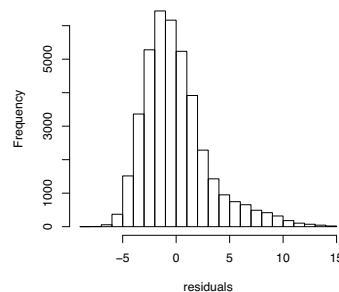


Figure 4: Residual distribution of performance model.

Messenger servers, over a period of 45 days. To keep the data collection process lightweight, aggregated values over 1 second are recorded every 30 seconds.

Given time series data of all available variables, we intend to identify important variables affecting the response variable (CPU utilization) and build a model of the response variable. Our statistical modeling methodology includes various exploratory data analysis, bi-direction model selection, diagnostic procedures and performance validation. Readers can refer to our technical report [4] for details.

The final model obtained from our methodology is:

$$\hat{U} = 2.84 \times 10^{-4} \cdot N + 0.549 \cdot L - 0.820 \quad (1)$$

where \hat{U} denotes the estimate of the CPU utilization percentage—the conditional mean in linear models, N is the number of active connections, and L is login rate. For example, given 70,000 connections and 15 new logins per second, the model estimates the CPU utilization to be 27.3%. The residual (error) distribution of this model can be seen from Figure 4. It shows that for 91.4% percent of cases, the observed values are within the range from -5 to 5 of the estimates from the model. Referring to the previous example, the actual CPU utilization falls in between 22.30 and 32.30 with probability 0.914. About 7.6% cases for which the model makes underestimation have larger residuals from 5 to 15. The remaining 1% cases have residuals less than -5 .

We use 2 weeks of data to train the model, and another 4 weeks for validation. The validation demonstrates that the distributions of testing errors are consistent with the distribution of training errors, concluding that CPU usage (and hence power consumption) of Messenger servers can reasonably be modeled with login rate (L) and number of active connections (N). Moreover, for a given maximum tolerable CPU utilization (defined by the Messenger operations group), the model also provides N_{\max} and L_{\max} , maximum tolerable values for N and L . Our provisioning and load dispatching algorithms therefore consider only these two variables and try to ensure that $N < N_{\max}$ and $L < L_{\max}$, while minimizing total energy consumption.

4 Energy-Aware Server Provisioning

As we explained in the previous section, there are two aspects of the connection load, namely the login rate and number of connections. Performance modeling in Section 3.5 validates the operation practice of setting limits L_{\max} and N_{\max} on individual servers for purposes of server provisioning and load dispatching. In this section, we discuss server provisioning methods that take into account of these two hard limits.

Let $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$ denote the total login rate and total number of connections at any given time t . Ideally we would calculate the number of servers needed as

$$K(t) = \max \left\{ \left\lceil \frac{L_{\text{tot}}(t)}{L_{\max}} \right\rceil, \left\lceil \frac{N_{\text{tot}}(t)}{N_{\max}} \right\rceil \right\}. \quad (2)$$

Here the notation $\lceil x \rceil$ denotes the smallest integer that is larger than or equal to x . The number $K(t)$ is calculated on a regular basis, for example, every half an hour.

There are two problems with this simple formula. First, $K(t)$ usually needs to be calculated ahead of time based on forecasted values of $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$, which can be inaccurate. This is especially the case if we need to turn on (cool start) servers to accommodate anticipated load increase. The lead time from starting the server to getting it ready is considered significant, with new logins arriving at a fast rate. Waking up servers in stand-by mode takes less time, but uncertainty still exists because of short-period fluctuation of the load.

Another problem of this simple calculation is less obvious but more critical. Using Equation (2) assumes that we can easily dispatch load to fill each server's capacity, say, almost instantaneously. However, this is not the case here because of the dynamic relationship between login rate and number of connections. For example, when a new server is turned on, we cannot expect it to take the full capacity of N_{\max} connections in short time. The number of connections on a server can only increase gradually, constrained by the bound L_{\max} on login rate. (For Messenger connection servers, it takes more than an hour to fill a server from empty.) The dynamic behavior of the system, when coupled with a load dispatching algorithm, requires additional margin in calculating $K(t)$.

A natural idea to fix both problems is to add extra margins in server provisioning:

$$K(t) = \max \left\{ \left\lceil \gamma_L \frac{L_{\text{tot}}(t)}{L_{\max}} \right\rceil, \left\lceil \gamma_N \frac{N_{\text{tot}}(t)}{N_{\max}} \right\rceil \right\}, \quad (3)$$

where the multiplicative factors satisfy $\gamma_L > 1$ and $\gamma_N > 1$. It remains the problem of how to determine these two factors. If they are chosen too big, we have over provisioning, which leads to inefficiency in saving energy; if they are chosen too small, we end up with under provisioning, which compromises quality of service.

Choosing the right margin factors requires careful evaluation of the forecasting accuracy, as well as detailed analysis of the dynamic behavior of the load dispatching algorithm. We will split the factors as

$$\gamma_L = \gamma_L^{\text{frc}} \gamma_L^{\text{dyn}}, \quad \gamma_N = \gamma_N^{\text{frc}} \gamma_N^{\text{dyn}}$$

where the superscript “frc” denotes forecasting factors that are used to compensate for forecasting errors; the superscript “dyn” denotes dynamic factors to compensate for dynamic behaviors caused by the load dispatching algorithm used. In the rest of this section, we focus on determining the forecasting factors. The dynamic factors will be discussed in detail in Section 5, along with the description of different load dispatching algorithms.

4.1 Hysteresis-based provisioning

We first consider a simple provisioning method based on hysteresis switching, without explicit forecasting.

At the beginning of each scheduling interval, say time t , we need to calculate $K(t+1)$, the number of servers that will be needed at time $t+1$, and schedule servers to be turned on or off accordingly. The information we have at time t are the observed values of $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$. Using Equation (2) (or (3), but only with the dynamic factors), we can estimate the number of servers needed at time t . Call this estimated number $\hat{K}(t)$. By comparing it with the actual number of active servers $K(t)$, we determine $K(t+1)$ as follows:

$$K(t+1) = \begin{cases} K(t), & \text{if } \gamma_{\text{low}} \hat{K}(t) \leq K(t) \leq \gamma_{\text{high}} \hat{K}(t) \\ \left\lceil \frac{1}{2} (\gamma_{\text{low}} + \gamma_{\text{high}}) \hat{K}(t) \right\rceil, & \text{otherwise.} \end{cases}$$

Here the two hysteresis margin factors γ_{low} and γ_{high} satisfy $\gamma_{\text{high}} > \gamma_{\text{low}} > 1$.

This method does not use load forecasting explicitly. However, in order to choose appropriate values for the two hysteresis parameters γ_{low} and γ_{high} , we need to have good estimate of how fast the load ramps up and down based on historical data. This method is especially useful when the load variations are hard to predict, for example, when special events happen or for holidays that we do not have enough historical data to give accurate forecast.

4.2 Forecast-based provisioning

The load of connection servers demonstrates a so-called seasonal characteristic that is common for many Internet services, electrical power networks, and many economic time series. In particular, it has periodic components in days, weeks, and even months, as well as a long-term growth trend. For the purpose of server provisioning, it

suffices to consider short-term load forecasting — forecasting over a period from half an hour to several hours. (Midterm forecasting is for days and weeks, and long-term forecasting is for months and years.)

There is extensive literature on short-term load forecasting of seasonal time series (see, e.g., [12, 2]). We derive a very simple and intuitive algorithm in Section 4.3, which works extremely well for the connection server data. To the best of our knowledge, it is a new addition to the literature of short-term load forecasting. On the other hand, however, the following discussion applies no matter which forecasting algorithm is used, as long as its associated forecast factors are determined, as we will do in Section 4.3 for our algorithm.

If the forecasting algorithm anticipates increased load $L_{\text{tot}}(t+1)$ and $N_{\text{tot}}(t+1)$ at time $t+1$, we can simply determine $K(t+1)$, the number of servers needed, using equation (3). The only thing we need to make sure is that new servers are turned on early enough to take the increased load. This usually is not a problem, for example, if we do forecasting over half an hour into the future.

More subtleties are involved in turning off servers. If the forecasted load will decrease, we will need less number of servers. Simply turning off one or more servers that are fully loaded will cause a sudden burst of SIDs. When disconnected users try to re-login at almost the same time, an artificial surge of login requests is created, which will stress the remaining active servers. When the new login requests on the remaining servers exceed L_{max} , SNA errors will be generated.

A better alternative is to schedule draining before turning a server off. More specifically, the dispatcher identifies servers that have the least amount of connections, and schedules them to connect to other servers at a controlled much slower pace that will not generate any significant burden for remaining active servers.

In order to reduce the number of SIDs, we can also starve the servers (simply not feeding it any new logins) for a period of time before doing scheduled draining or shutting it down. For Messenger servers, the natural departure rate caused by normal user logoffs results in an exponential decay of the number of connections, with a time constant slightly less than an hour, meaning that the number of connections on a server decreases by half every hour. A two-hour starving time leads to number of SIDs less than a quarter of that without starving. The trade-off is that adding starving time reduces efficiency in saving energy.

4.3 Short-term load forecasting

Now we present our method for short-term load forecasting. Let $y(t)$ be the stochastic periodic time series under consideration, with a specified time unit. It can repre-

sent $L_{\text{tot}}(t)$ or $N_{\text{tot}}(t)$ measured at regular time intervals. Suppose the periodic component has a period of T time units. We express the value of $y(t)$ in terms of all previous measurements as

$$y(t) = \sum_{k=1}^n a_k y(t - kT) + \sum_{j=1}^m b_j \Delta y(t - j),$$

$$\Delta y(t - j) = y(t - j) - \frac{1}{n} \sum_{k=1}^n y(t - j - kT).$$

There are two parts in the above model. The part with parameters a_k does periodic prediction — it is an autoregression model for the value of y over a period of T . The assumption is that the values of y at every T steps are highly correlated. The part with parameters b_j gives local adjustment, meaning that we also consider correlations between $y(t)$ and the values immediately before it. The integers n and m are their orders, respectively. We call this a SPAR (Sparse Periodic Auto-Regression) model. It can be easily extended to one with multiple periodic components.

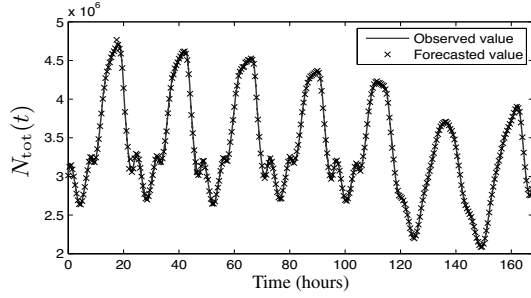
For the examples we will consider, short-term forecasting is done over half an hour, and we let the period T be a week, which leads to $T = 7 \times 24 \times 2 = 168$ samples. In this case, the autoregression part (with parameters a_k) of the SPAR model means, for example, the load at 9am this Tuesday is highly correlated and can be well predicted from the loads at 9am of previous Tuesdays. The local adjustment part (with parameters b_j) reflects the immediate trends predicted from values at 8:30am, 8:00am, and so on, on the same day.

We have tried several models with different orders n and m , but found that the coefficients a_k, b_j are very small for $k > 4$ and $j > 2$, and ignoring them does not reduce the forecasting accuracy by much. So we choose the orders $n = 4$ and $m = 2$ to do load forecasting in our examples. We used five weeks of data to estimate the parameters a_k and b_j (by solving a simple least-squares fitting problem). Then we use these data and estimated parameters to forecast loads for the following weeks.

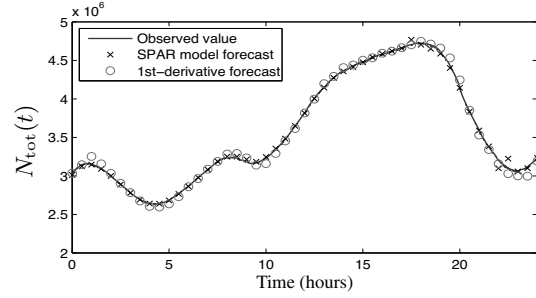
Figure 5 shows the results of using this forecasting model. The figures show the forecasted values (every 30 minutes) plotted against actually observations (measured every 30 seconds). Note that the observed login rates (measured every second and recorded every 30 seconds) appear to be very noisy and have lots of spikes. Using this model, we can reasonably forecast the smooth trend of the curve. The spikes are hard to predict, because they are mostly caused by irregular server unavailability or crashes that result in re-login bursts.

We computed the standard deviations of the relative errors $(\hat{L}(t) - L(t))/L(t)$ and $(\hat{N}(t) - N(t))/N(t)$:

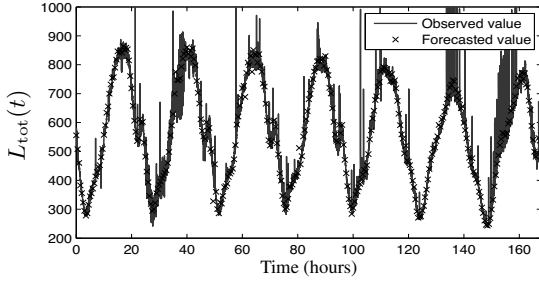
$$\sigma_L = 0.039, \quad \sigma_N = 0.006.$$



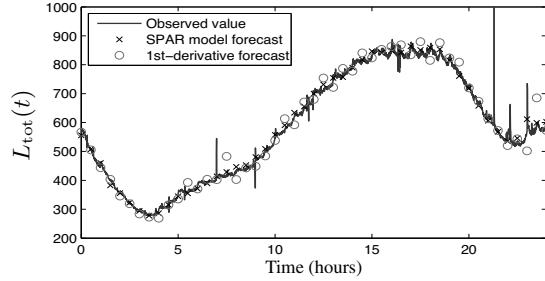
(a) Number of connections over a week.



(b) Number of connections on Monday.



(c) Login rates over a week.



(d) Login rates on Monday.

Figure 5: Short-term load forecasting.

We also compared it with some simple strategies for short-term load forecasting without using periodic time series models. For example, one idea is to predict the load at the next observation point based on the first-order derivative of the current load and the average load at the previous observation point (e.g., Heath et al. [14]). This approach leads to the standard deviations $\sigma_L = 0.079$ and $\sigma_N = 0.012$, which are much worse than the results of our SPAR model (see Figure 5 (b) and (d)). We expect that such simple heuristics would work well for smooth trend and very short forecasting intervals. Here our time series can be very noisy (especially the login rates), and a longer forecasting interval is preferred because we do not want to turn on and off servers too frequently. In particular, the SPAR model will work reasonably well for forecasting intervals of a couple of hours (and even longer), for which derivative-based heuristics will no longer make sense.

Given the standard deviations of the forecasting errors using the SPAR model, we can assign the forecasting factors as

$$\begin{aligned}\gamma_L^{\text{frc}} &= 1 + 3\sigma_L \approx 1.12, \\ \gamma_N^{\text{frc}} &= 1 + 3\sigma_N \approx 1.02.\end{aligned}\quad (4)$$

These forecast factors will be substituted into Equation (3) to determine the number of servers required. To do so, we also need to specify a load dispatching algorithm and its associated dynamic factors, which we explain in the next section.

5 Load Dispatching Algorithms

In this section, we present algorithms that decide how large a share of the incoming login requests should be given to each server. We describe two different types of algorithms — load balancing and load skewing, and determine their corresponding dynamic factors γ_L^{dyn} and γ_N^{dyn} . These two algorithms lead to different load distributions on the active servers, and have different implications in terms of energy saving and number of SIDs. In order to present them, we first need to establish a dynamic model of the load-dispatching system.

5.1 Dynamic system modeling

We consider a discrete-time model, where t denotes time with a specified unit. The time unit here is usually much smaller than the one used for load forecasting; for example, it is usually on the order of a few seconds. Let $K(t)$ be the number of active servers during the interval between time t and $t + 1$. Let $N_i(t)$ denote the number of connections on server i at time t , and $L_i(t)$ and $D_i(t)$ be the number of logins and departures, respectively, between time t and $t + 1$ (see Figure 5.1). The dynamics of the individual servers can be expressed as

$$N_i(t + 1) = N_i(t) + L_i(t) - D_i(t)$$

for $i = 1, \dots, K(t)$. This first-order difference equation captures the integration relationship between the login

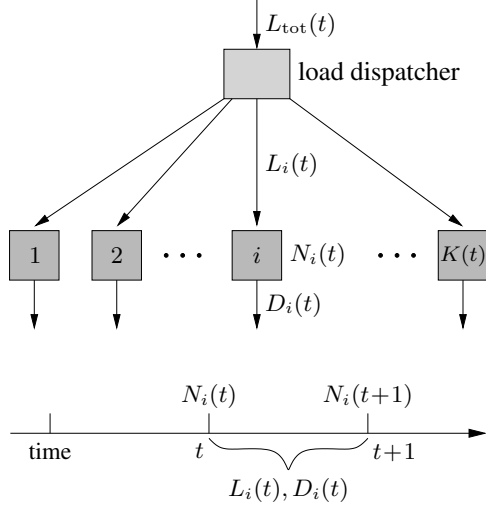


Figure 6: Load balancing of connection servers.

rates $L_i(t)$ and the number of connections $N_i(t)$. The number of departures $D_i(t)$ usually is a fraction of $N_i(t)$, which varies a lot from time to time.

The job of the dispatcher is to dispatch the total incoming login request, $L_{tot}(t)$, to the available $K(t)$ servers. In other words, it determines $L_i(t)$ for each server i . In general, a dispatching algorithm can be expressed as

$$L_i(t) = L_{tot}(t)p_i(t), \quad i = 1, \dots, K(t)$$

where $p_i(t)$ is the portion or fraction of the total login requests assigned to the server i (with $\sum_i p_i(t) = 1$). For a randomized algorithm, $p_i(t)$ stands for the probabilities with which the dispatcher distributes the load.

5.2 Load balancing

Load balancing algorithms try to make the numbers of connections on the servers the same, or as close as possible. The simple method of round-Robin load-balancing, i.e., always letting $p_i(t) = 1/K(t)$, apparently does not work here. By setting a uniform login rates for all the servers, regardless of the fluctuations of departure rates on individual servers (an open-loop strategy), it leaves the number of connections on the servers diverge without proper feedback control.

There are many ways to make load balancing work for such a system. For example, one effective heuristic is to apply round-Robin only to a fraction of the servers that have relatively small number of connections. In this paper we describe a proportional load-balancing algorithm, where the dispatcher assigns the following portion of total loads to server i :

$$p_i(t) = \frac{1}{K(t)} + \alpha \left(\frac{1}{K(t)} - \frac{N_i(t)}{N_{tot}(t)} \right) \quad (5)$$

where $\alpha > 0$ is a parameter that can be tuned to influence the dynamic behavior of the system. Intuitively, this algorithm assigns larger portions to servers with relatively small number of connections, and smaller portions to servers with relatively large number of connections. Using the fact $N_{tot}(t) = \sum_{i=1}^{K(t)} N_i(t)$, we always have $\sum_{i=1}^{K(t)} p_i(t) = 1$. However, we notice that $p_i(t)$ can be negative. In this case, the dispatcher actually take load off from server i instead of assigning new load to it. This can be done by either migrating connections internally, or by going through the loop of disconnecting some users and automatic re-login onto other servers.

This algorithm leads to a very interesting property of the system: every server has the same closed-loop dynamics, only with different initial conditions. All the servers will behave exactly the same as time goes on. The only exceptions are the newly turned-on servers which have zero initial number of connections. Turning off servers does not affect others if the load are reconnected according to the same proportional rule in Equation (5). Detailed analysis of the properties of this algorithm is given in [4].

Now let's examine carefully the effect of the parameter α . For small values of α , the algorithm maintains relatively uniform login rates to all the servers, so it can be very slow in driving the number of connections to uniform. The extreme case of $\alpha = 0$ (which is disallowed here) would correspond to round-Robin. For large values of α , the algorithm tries to drive the number of connections quickly to uniform, by relying on disparate login rates across servers. In terms of determining γ_L^{dyn} , we note that the highest login rate is always assigned to newly turned-on servers with $N_i(t) = 0$. In this case, $p_i(t) = (1 + \alpha)/K(t)$. By requiring $p_i(t)L_{tot}(t) \leq L_{max}$, we obtain $K(t) \geq (1 + \alpha)L_{tot}(t)/L_{max}$. Comparing with Equation (3), we have $\gamma_L^{dyn} = 1 + \alpha$.

The determination of γ_N^{dyn} is more involved. The details are omitted due to space constraints (but can be found in [4]). Here we simply list the two factors:

$$\gamma_L^{dyn} = 1 + \alpha, \quad \gamma_N^{dyn} = \frac{1 + \alpha}{r + \alpha} \quad (6)$$

where $r = \min_t D_{tot}(t)/L_{tot}(t)$, the minimum ratio among any time t between $D_{tot}(t)$ and $L_{tot}(t)$ (the total departures and total logins between time t and $t + 1$). In practice, r is estimated based on historical data.

In summary, tuning the parameter α allows us to trade-off the two terms appearing in the formula (3) for determining number of servers needed. Ideally, we shall choose an α that makes the two terms approximately equal for typical ranges of $L_{tot}(t)$ and $N_{tot}(t)$, that is, makes the constraints tight simultaneously for both maximum login rate and maximum number of connections.

5.3 Load skewing

The principle of load skewing is exactly the opposite of load balancing. Here new login requests are routed to busy servers as long as the servers can handle them. The goal is to maintain a small number of tail servers that have small number of connections. When user login requests ramp up, these servers will be used as reserve to handle login increases and surge, and give time for new servers to be turned on. When user login requests ramp down, these servers can be slowly drained and shut down. Since only tail servers are shut down, the number of SIDs can be greatly reduced, and no artificial surge of re-login requests or connection migrations will be created.

There are many possibilities to do load skewing. Here we describe a very simple scheme. In addition to the hard bound N_{\max} on the number of connections a server can take, we specify a target number of connections N_{tgt} , which is slightly smaller than N_{\max} . When dispatching new login requests, servers with loads that are smaller than N_{tgt} and closest to N_{tgt} are given priority. Once a server's number of connections reaches N_{tgt} , it will not be assigned new connections for a while, until it drops again below N_{tgt} due to gradual user departures.

More specifically, let $0 < \rho < 1$ be a give parameter. At each time t , the dispatcher always distributes new connections evenly (round-Robin) to a fraction ρ of all the available servers. Let $K(t)$ be the number of servers available, it will choose the $\lceil \rho K(t) \rceil$ servers in the following way. First, the dispatcher partitions the set of servers $\{1, 2, \dots, K(t)\}$ into two subsets:

$$\begin{aligned} I_{\text{low}}(t) &= \{i \mid N_i(t) < N_{\text{tgt}}\} \\ I_{\text{high}}(t) &= \{i \mid N_i(t) \geq N_{\text{tgt}}\} \end{aligned}$$

Then it chooses the top $\lceil \rho K(t) \rceil$ servers (those with the highest number of connections) in $I_{\text{low}}(t)$. If the number of servers in $I_{\text{low}}(t)$ is less than $\lceil \rho K(t) \rceil$, the dispatcher has two choices. It can either distribute load evenly only to servers in $I_{\text{low}}(t)$, or it can include the bottom $\lceil \rho K(t) \rceil - |I_{\text{low}}(t)|$ servers in $I_{\text{high}}(t)$. In the second case, the number N_{tgt} is set further away from N_{\max} to avoid number of connections exceeding N_{\max} (i.e., SNA errors) within a short time.

This algorithm will lead to a skewed load distribution across the available servers. Most of the active servers should have number of connections close to N_{tgt} , except a small number of tail servers. Let the desired number of tail servers be K_{tail} . The dynamic factors for this algorithm can be easily determined as

$$\gamma_L^{\text{dyn}} = \frac{1}{\rho}, \quad \gamma_N^{\text{dyn}} = 1 + \frac{K_{\text{tail}}}{\min_t N_{\text{tot}}(t)/N_{\text{tgt}}}. \quad (7)$$

These factors can be substituted into equation (3) to calculate the number of servers needed $K(t)$, where it can

be combined with either hysteresis-based or forecast-based server provisioning.

5.3.1 Reactive load skewing

The load skewing algorithm is especially suitable to reduce the number of SIDs when turning off servers. To best utilize load skewing, we also develop a heuristic called reactive load skewing (RLS). In particular, it is a hysteresis rule to control the number of tail servers. For this purpose, we need to specify another number N_{tail} . Servers with number of connections less than N_{tail} are called tail servers. Let $K_{\text{tail}}(t)$ be the number of tail servers, and $K_{\text{low}} < K_{\text{high}}$ be two thresholds. If $K_{\text{tail}}(t) < K_{\text{low}}$, then $\lceil (K_{\text{high}} - K_{\text{low}})/2 \rceil - K_{\text{tail}}(t)$ servers are turned on. If $K_{\text{tail}}(t) > K_{\text{high}}$, then $K_{\text{tail}}(t) - K_{\text{high}}$ servers are turned off. The tail servers have very low active connections, so turning off one or even several of them will not create artificial reconnection spike. This on-off policy is executed at the server provisioning time scale, for example, every half an hour.

6 Evaluations

In this section, we compare the performance of different provisioning and load dispatching algorithms through simulations based on real traffic traces.

6.1 Experimental setup

We simulate a cluster of 60 connection servers with real data traces of total connected users and login rates obtained from production Messenger connection servers (as described in Section 3.5). The data traces are scaled accordingly to fit on 60 servers; see Figure 2. These 60 servers are treated as one cluster with a single dispatcher.

The server power model is measured on an HP server with two dual-core 2.88GHz Xeon processors and 4G memory running Windows Server 2003. We approximate server power consumption (in Watts) as

$$P = \begin{cases} 150 + 0.75 \times U, & \text{if active} \\ 3, & \text{if stand-by} \end{cases} \quad (8)$$

where U is the CPU utilization percentage from 0 to 100 (see Figure 3). The CPU utilization is modeled using the relationship derived in Section 3.5, in particular equation (1). CPU utilization is bounded within 5 to 100 percent when the server is active.

The limits on connected users and login rate are set by Messenger stress testing:

$$N_{\max} = 100,000, \quad L_{\max} = 70/\text{sec}.$$

Also through testing, the server's wake-up delay, defined as the duration from a message is sent to wake up the server till the server successfully joins the cluster, is 2

minutes. The draining speed, defined as the number of users disconnected once a server decide to shut down, is 100 connections per second. This implies that it takes about 15 minutes to drain a fully loaded server.

6.2 Forecast vs. hysteresis provisioning

We first compare the performance of no provisioning, forecast-based and hysteresis-based provisioning, with a common load balancing algorithm.

- **No provisioning with load balancing (NB).** NB uses all the 60 servers. It implements the load balancing algorithm in Section 5.2 with $\alpha = 1$.
- **Forecast provisioning with load balancing (FB).** FB uses the load balancing algorithm in Section 5.2 with $\alpha = 1$ and the load forecasting algorithm in Section 4.3. The number of servers $K(t)$ are calculated using equation (3) with the factors

$$\begin{aligned} \gamma_L^{\text{frc}} &= 1.12, \quad \gamma_N^{\text{frc}} = 1.02 && \text{from equation (4),} \\ \gamma_L^{\text{dyn}} &= 2, \quad \gamma_N^{\text{dyn}} = 1.05 && \text{from equation (6).} \end{aligned}$$

In calculating γ_N^{dyn} , we used the estimation $r = 0.9$ obtained from historical data.

- **Hysteresis provisioning with load balancing (HB).** HB uses the same load balancing algorithm, but with the hysteresis-based provisioning method in Section 4.1. In calculating $K(t)$, it uses the same dynamic factors as FB. There is no forecast factors for HB. Instead, we tried three pairs of hysteresis margins $(\gamma_{\text{low}}, \gamma_{\text{high}})$:

$$(1.05, 1.10), \quad (1.04, 1.08), \quad (1.04, 1.06).$$

denoted as HB(5/10), HB(4/8) and HB(4/6).

The simulation results based on two days of real data (Monday and Tuesday in Figure 2) are listed in Table 1. The number of servers used are shown in Figure 7. These results show that forecast-based provisioning leads to slightly more energy savings than hysteresis-based provisioning. With the hysteresis margins getting tight, the difference in energy savings becomes even smaller. However, this comes with a cost of service quality degradation, as shown by the increased numbers of SNA errors caused by smaller hysteresis margins.

Algorithm	Energy (kWh)	Saving	SNA
NB ($\alpha = 1$)	478	—	0
FB ($\alpha = 1$)	331	30.8%	0
HB(5/10)	344	28.0%	0
HB(4/8)	341	28.6%	7,602
HB(4/6)	338	29.2%	512,531

Table 1: Comparison of provisioning methods. Energy savings are reduced percentages with respect to NB.

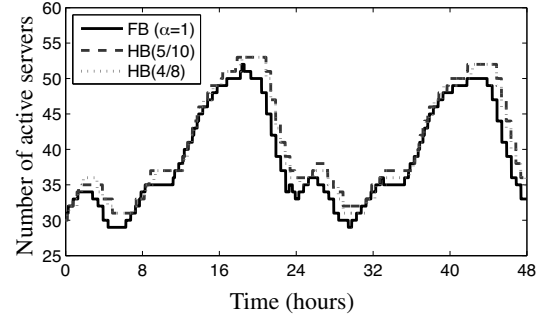


Figure 7: Number of servers used by FB and HB.

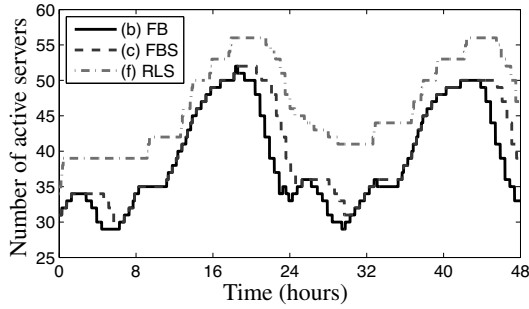
6.3 Load balancing vs. load skewing

In addition to energy saving, the number of SIDs is another important performance metric. We evaluate both aspects on FB and the following algorithms:

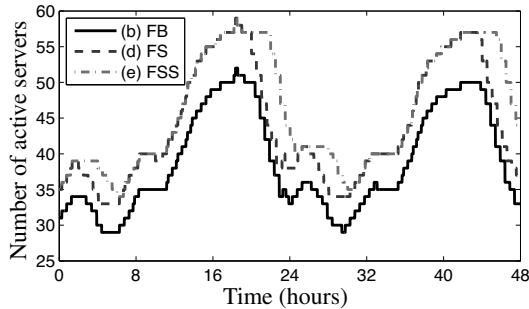
- **Forecast provisioning with load balancing and Starving (FBS).** FBS uses the same parameters as FB, with the addition of a period of starving time before turning off servers. We denote the starving time by S . For example, $S = 2$ means starving for two hours before turning off.
- **Forecast provisioning with load skewing (FS).** FS uses the same forecast provisioning method as before, combined with the load skewing algorithm in Section 5.3 with parameters $\rho = 1/2$ and $N_{\text{tgt}} = 98,000$. Its associated dynamic factors are obtained from Equation (7): $\gamma_L^{\text{dyn}} = 2$ and $\gamma_N^{\text{dyn}} = 1.2$, where $K_{\text{tail}} = 6$ is used in calculating γ_N^{dyn} .
- **Forecast provisioning with load skewing and starving (FSS).** FSS uses the same algorithms and parameters as FS, with the addition of a starving time S hours before turning off servers.
- **Reactive load skewing (RLS).** RLS uses the load skewing algorithm with the same ρ and N_{tgt} as FS. Instead of load forecasting, it uses the hysteresis on-off scheme in Section 5.3.1 with parameters $N_{\text{tail}} = N_{\text{max}}/10 = 10,000$, $K_{\text{low}} = 2$ and $K_{\text{high}} = 6$.

Simulation results of some typical scenarios, labeled as (a),(b),(c),(d),(e),(f), are shown in Table 2 and Figure 8. Comparing FBS with FB and FSS with FS, we see that adding a two-hour starving time before turning off servers leads to significant reduction in the number of SIDs, and mild increase in energy consumption.

The results in Table 2 show a clear tradeoff between energy consumption and number of SIDs. With the same amount of starving time, load balancing uses less energy but generates more SIDs, and load skewing uses more energy but generates less SIDs. In particular, load skewing without starving has less number of SIDs than load



(a) Number of servers used by FB, FBS and RLS.



(b) Number of servers used by FB, FS and FSS.

Figure 8: Number of servers by different algorithms.

balancing with starving for two hours. RLS with a relatively small N_{tail} (say around 10,000) has less number of SIDs even without starving. To give a better perspective of the SID numbers, we note that the total number of logins during these two days is over 100 millions (again, this is the number scaled to 60 servers).

6.3.1 Load profiles

To give more insight into different algorithms, we show their load profiles in Figure 9. Each vertical cut through the figures represents the load distribution (sorted number of connections) across the 60 servers at a particular time. For NB, all the loads are evenly distributed on the 60 servers, so each vertical cut has uniform load distribution, and they vary together to follow the total load pattern. Other algorithms use server provisioning to save energy, so each vertical cut has a drop to zero at the number of servers they use. The contours of different number of connections are plotted. The highest contour curves show the numbers of servers used (those with nonzero connections), cf. Figure 8.

For FB, the contour lines are very dense (sudden drop to zero), especially when the total load is ramping down and servers need to be turned off. This means servers with almost full loads have to be turned off, which causes lots of SIDs. Adding starving time makes the contour lines of FBS sparser, which corresponds to reduced num-

	Algorithms and Parameters	Energy (kWh)	Saving	Number of SIDs
(a)	NB	478	—	0
(b)	FB ($\alpha = 1$)	331	30.8%	3,711,680
(c)	FBS ($\alpha = 1, S=2$)	343	28.2%	799,120
(d)	FS ($\rho = 0.5$)	367	23.3%	597,520
(e)	FSS ($\rho = 0.5, S=2$)	381	20.2%	115,360
(f)	RLS ($\rho = 0.5, N_{\text{tail}} = 10,000$)	375	21.5%	48,160

Table 2: Comparison of load dispatching algorithms. All algorithms in this table have zero SNA errors.

ber of SIDs. Load skewing algorithms (FS, FSS and RLS) intentionally create tail servers, which makes the contour lines much sparser. While they consume a bit more energy, the number of SIDs are dramatically reduced by turning off only tail servers.

6.3.2 Energy-SID tradeoffs

To unveil the complete picture of the energy-SID tradeoffs, we did extensive simulation of different algorithms by varying their key parameters.

Not all possible parameter variations give meaningful results. For example, if we choose ρ too small for FSS or RLS (e.g., $\rho \leq 0.4$ for this particular data trace), significant amount of SNA errors will occur because small ρ limits the cluster's capability of taking high login rates. The number of SNA errors will also increase significantly if we set $N_{\text{tail}} \geq 40,000$ in RLS. For fair comparison, all scenarios shown in Figure 10 give less than 1000 SNA errors (due to rare spikes in login rates).

For FBS, the three curves correspond to the parameters $\alpha = 0.2, 1.0$, and 3.0 . Each curve is generated by varying the starving time S from 0 to 8 hours, evaluated at every half-an-hour increment (labeled as crosses). Within the figure, it only shows parts of the curves to allow better visualization of other curves. The right-most crosses on each curve correspond to $S = 2$ hours, and the number of SIDs decreases as S increases.

For FSS, the three curves correspond to the parameters $\rho = 0.4, 0.5, 0.9$. Each curve is generated by varying the starving time S from 0 to 5 hours, evaluated at every half an hour (labeled as triangles). For example, scenario (d) in Table 2 is the right-most symbol on the curve $\rho = 0.5$. The plots for FBS and FSS show that the number of SIDs roughly decreases by half for every hour of starving time (every two symbols on the curves).

For RLS, the three curves correspond to the parameters $\rho = 0.4, 0.5$, and 0.6 . Each curve is generated by varying the threshold for tail servers N_{tail} from 1000 to 40,000 (labeled as circles). The number of SIDs increases as N_{tail} increases (the right-most circles are for

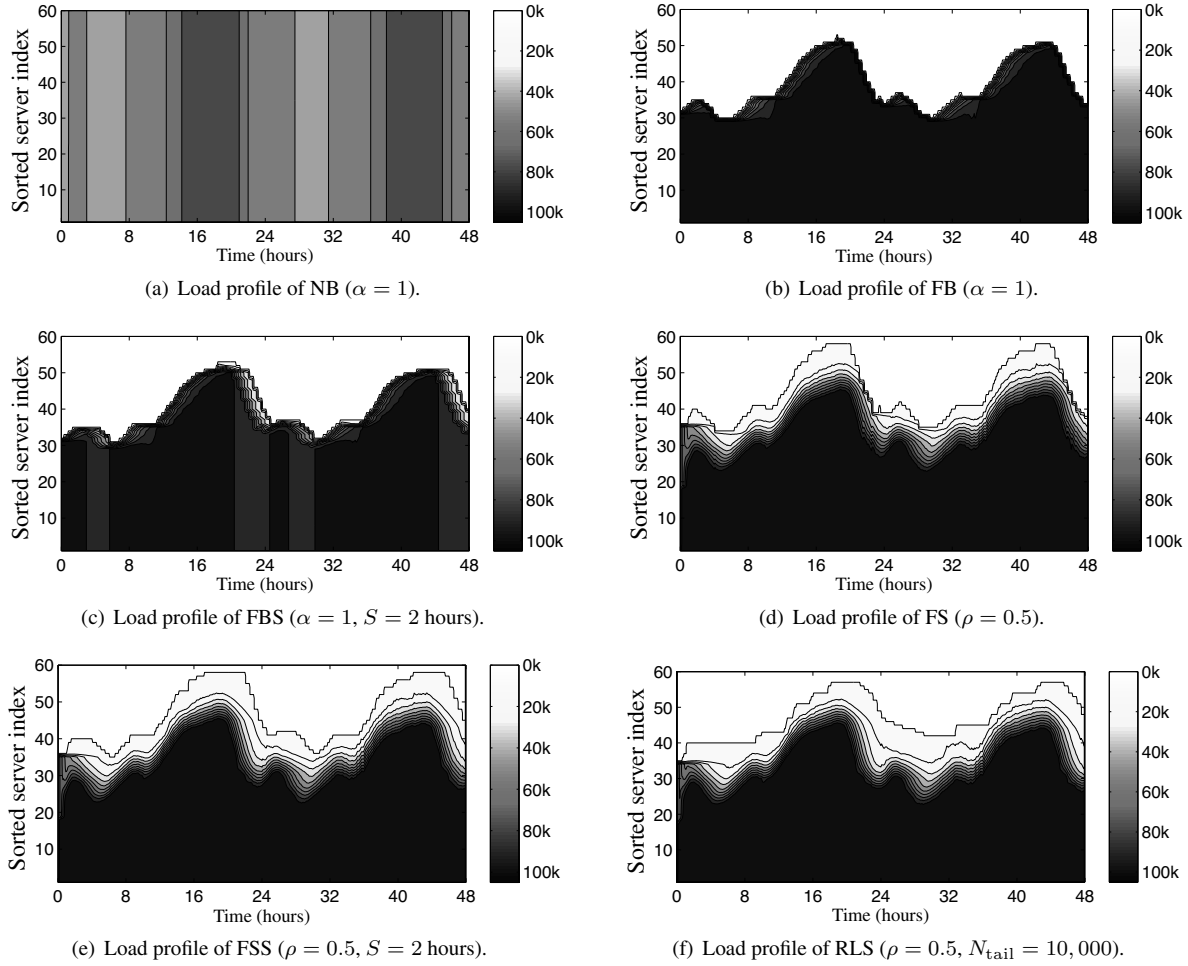


Figure 9: Load profile of different server scheduling and load dispatching algorithms shown in Table 2.

$N_{\text{tail}}=40,000$). Decreasing the threshold N_{tail} for RLS is roughly equivalent to increasing starving time S for FBS and FSS.

In Figure 10, points near the bottom-left corner have the desired property of both low energy and low number of SIDs. In this perspective, FSS can be completely dominated by both FBS and RLS *if their parameters are tuned appropriately*. For FBS, it takes a much longer starving time than FSS to reach the same number of SIDs, but the energy consumption can be much less if the value of α is chosen around 1.0. Between FBS and RLS, they have their own sweet spots of operation. For this particular data trace, FBS with $\alpha = 1$ and 4 to 5 hours of starving time give excellent energy-SID tradeoff.

7 Discussions

From the simulation results in Section 6, we see that while load skewing algorithms generate small number of SIDs when turning off servers, they also maintain unnec-

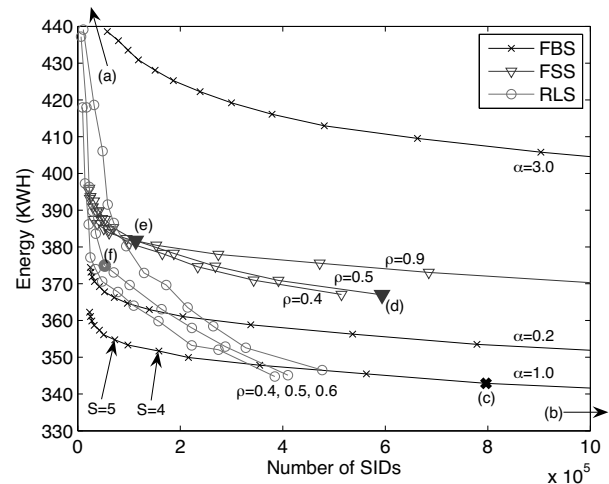


Figure 10: Energy-SID tradeoff of different algorithms with key parameters varied. The particular scenarios (c),(d),(e),(f) listed in Table 2 are labeled with bold symbols. The scenarios (a) and (b) are out of scope of this figure, but their relative locations are pointed by arrows.

essary tail servers when the load ramps up. So a hybrid (switching) algorithm that employs load balancing when load increases and load-skewing when load decreases seems to be able to get the best energy-SID tradeoff. This is what the FBS algorithm tries to do with a long starving time, and its effectiveness is clearly seen in Figure 10. Of course, there are regions in the energy-SID tradeoff plane that favor RLS more than FBS. This indicates that there are still room to improve by explicitly switching between load balancing and load skewing, for example, when the total number of connections reaches the peak and starts to go down.

As mentioned in Section 3.4, SIDs can be handled transparently by the client software so that they are unnoticeable to users, and the servers can be scheduled to drain slowly in order to avoid creating a surge of reconnection requests. The transitions can also be done through “controlled connection migration” (CCM) — to migrate the TCP connection endpoint state without breaking it. Depending on the implementation details, CCM may also be a CPU- or networking-intensive activity. The number of CCMs could be a performance metric similar to the number of SIDs, which we trade off with energy saving. Depending on the cost of CCMs, they might change the sweet spots on the energy-SID/CCM tradeoff plane. If the cost is low, due to a perfect connection migration scheme, then we can be more aggressive on energy saving. On the other hand, if the cost is high, we have to be more conservative, as in dealing with SIDs. We believe the analysis and algorithmic framework we developed would still apply.

We end the discussions with some practical considerations of implementing the dynamic provisioning strategy at full scale. Implementing the core algorithms is relatively straightforward. However, we need to add some safeguarding outer loops to ensure they work in their comfort zone. For example, if the load-forecasting algorithm starts giving large prediction errors (e.g., when there is not enough historical data to produce accurate model for special periods such as holidays), we need to switch to the more reactive hysteresis-based provisioning algorithm. To take the best advantage of both load balancing and load skewing, we need a simple yet robust mechanism to detect the up and down trends in the total load pattern. On the hardware side, frequently turning on and off servers may raise reliability concern. We want to avoid always turning on and off the same machines. This is where load prediction can help by looking ahead and avoiding short term decisions. A better solution is to rotate servers in and out of the active clusters deliberately.

Acknowledgments

We thank the Messenger team for their support, and Jeremy Elson and Jon Howell for helpful discussions.

References

- [1] ABDELZAHER, T. F., SHIN, K. G., AND BHATTI, N. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 1 (2002), 80–96.
- [2] BROCKWELL, P. J., AND DAVIS, R. A. *Introduction to Time Series and Forecasting*, second ed. Springer, 2002.
- [3] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. Managing Energy and Server Resources in Hosting Centers. In *SOSP* (2001).
- [4] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. Tech. rep., Microsoft Research, 2007.
- [5] CHEN, Y., DAS, A., QIN, W., SIVASUBRAMANIAM, A., WANG, Q., AND GAUTAM, N. Managing server energy and operational costs in hosting centers. In *In Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (2005).
- [6] DOYLE, R., CHASE, J., ASAD, O., JIN, W., AND VAHDAT, A. Model-Based Resource Provisioning in a Web Service Utility. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003).
- [7] ELNOZAHY, M., KISTLER, M., AND RAJAMONY, R. Energy conservation policies for web servers. In *USITS* (2003).
- [8] EPA Report on Server and Data Center Energy Efficiency. U.S. Environmental Protection Agency, ENERGY STAR Program, 2007.
- [9] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power Provisioning for a Warehouse-sized Computer. In *ISCA* (2007).
- [10] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic Performance-Setting for Linux. In *OSDI* (2002).
- [11] GANESH, L., WEATHERSPOON, H., BALAKRISHNAN, M., AND BIRMAN, K. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)* (2007).
- [12] GROSS, G., AND GALIANA, F. D. Short-term load forecasting. *Proceedings of The IEEE* 75, 12 (1987), 1558–1573.
- [13] GRUNWALD, D., LEVIS, P., FARKAS, K. I., III, C. B. M., AND NEUFELD, M. Policies for Dynamic Clock Scheduling. In *OSDI* (2000).
- [14] HEATH, T., DINIZ, B., CARRERA, E. V., JR., W. M., AND BIANCHINI, R. Energy conservation in heterogeneous server clusters. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2005).
- [15] KANDASAMY, N., ABDELWAHED, S., AND HAYES, J. P. Self-optimization in computer systems via online control: Application to power management. In *IEEE International Conference on Autonomic Computing ICAC* (2004).
- [16] KUSIC, D., AND KANDASAMY, N. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *IEEE International Conference on Autonomic Computing ICAC* (2006).
- [17] LORCH, J. R., AND SMITH, A. J. Improving Dynamic Voltage Scaling Algorithms with PACE. In *SIGMETRICS/Performance* (2001).
- [18] MSN protocol documentation. <http://msnpiki.msnfanatic.com/>.
- [19] M. WEISER, B. WELCH, DEMERS, A. J., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *OSDI* (1994).
- [20] PINHEIRO, E., BIANCHINI, R., CARRERA, E. V., AND HEATH, T. Dynamic Cluster Reconfiguration for Power and Performance. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power* (2001).
- [21] SHARMA, V., THOMAS, A., ABDELZAHER, T., SKADRON, K., AND LU, Z. Power aware QoS management in web servers. In *IEEE RTSS* (2003).
- [22] WANG, M., KANDASAMY, N., GUEZ, A., AND KAM, M. Distributed cooperative control for adaptive performance management. *IEEE Internet Computing* 11, 1 (2007), 31–39.
- [23] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: Helping Disk Arrays Sleep Through the Winter. In *SOSP* (2005).

Consensus Routing: The Internet as a Distributed System

John P. John* Ethan Katz-Bassett* Arvind Krishnamurthy* Thomas Anderson*

Arun Venkataramani†

Abstract

Internet routing protocols (BGP, OSPF, RIP) have traditionally favored responsiveness over consistency. A router applies a received update immediately to its forwarding table before propagating the update to other routers, including those that potentially depend upon the outcome of the update. Responsiveness comes at the cost of routing loops and blackholes—a router A thinks its route to a destination is via B but B disagrees. By favoring responsiveness (a liveness property) over consistency (a safety property), Internet routing has lost both.

Our position is that consistent state in a distributed system makes its behavior more predictable and securable. To this end, we present consensus routing, a consistency-first approach that cleanly separates safety and liveness using two logically distinct modes of packet delivery: a *stable* mode where a route is adopted only after all dependent routers have agreed upon it, and a *transient* mode that heuristically forwards the small fraction of packets that encounter failed links. Somewhat surprisingly, we find that consensus routing improves overall availability when used in conjunction with existing transient mode heuristics such as backup paths, deflections, or detouring. Experiments on the Internet's AS-level topology show that consensus routing eliminates nearly all transient disconnection in BGP.

1 Introduction

Internet routing, especially interdomain routing, has traditionally favored responsiveness, i.e., how quickly the network reacts to changes, over consistency, i.e., ensuring that packets traverse adopted routes. A router applies a received update immediately to its forwarding table before propagating the update to other routers, including those that potentially depend upon the outcome of the update. Responsiveness comes at the cost of availability: a router A thinks its route to a destination is via B but B disagrees, either because 1) B's old route to the destination is via A, causing loops, or 2) B does not have a current route to the destination, causing blackholes. BGP updates are known to cause up to 30% packet-loss for two minutes or more after a routing change, even though usable physical routes exist [23]. Further, transient loops account for 90% of all packet loss according to a Sprint network study [14]. Even a recovering link can cause unavailability lasting

tens of seconds due to an inconsistent view among routers in a single autonomous system [38].

Our position is that the lack of consistency is at the root of bigger problems in Internet routing beyond availability. First, protocol behavior is complex and unpredictable as routers by design operate upon inconsistent distributed state, e.g., by forwarding packets along loops. There is no indicator of when, if at all, the network converges to a consistent state. Second, unpredictable behavior makes the system more vulnerable to misconfiguration or abuse, as it is difficult to distinguish between expected behavior and misbehavior. Third, unpredictable behavior stifles innovation in the long term, e.g., network operators are reluctant to adopt protocol optimizations such as interdomain traffic engineering [1] because they have to worry about its poorly understood side-effects. Perhaps most tellingly, despite a decade of research investigating the complex dynamics of interdomain routing, the *goal of a simple, practical routing protocol that allows general routing policies and achieves high availability* has remained elusive.

Our primary contribution, consensus routing, achieves the above goal. The key insight is to recognize consistency as a safety property and responsiveness as a liveness property and systematically separate the two design concerns, thereby borrowing an old lesson from distributed system design. Consistency safety means that a router forwards a packet strictly along the path adopted by the upstream routers unless the packet encounters a failed link. Liveness means that the system reacts quickly to failures or policy changes. Separating safety and liveness improves end-to-end availability, and, perhaps more importantly, makes system behavior simple to describe and understand.

Consensus routing achieves this separation using two logically distinct modes of packet delivery: 1) A *stable mode* ensures that a route is adopted only after all dependent routers have agreed upon a consistent view of global state. Every epoch, routers participate in a distributed snapshot and consensus protocol to determine whether or not updates are *complete*, i.e., they have been processed by every router that depends on the update. The output of the consensus serves as an explicit indicator that routers may adopt a consistent set of routes processed before the snapshot. 2) A *transient mode* ensures high availability when a packet encounters a router that does not possess a stable route, either because the corresponding link failed

*Dept. of Computer Science, Univ. of Washington, Seattle.

†University of Massachusetts Amherst.

or the consensus protocol to compute a stable route has not yet terminated. In this case, the router explicitly marks it as a transient packet, and uses local information about available routes heuristically to forward the packet to the destination. We show that consensus routing can cleanly accommodate a number of existing transient forwarding heuristics such as backup routes [21], deflections [41], and detours [40] to provide near-perfect availability in a policy-compliant manner.

Consensus routing is similar in spirit to recent work on intradomain routing protocols that advocate the separation of route computation from packet forwarding [24]. However, we address this challenge for interdomain routing where ASes can run *arbitrary and private policy* engines to select routes, thereby precluding the use of logically centralized schemes for route computation. Consensus routing needs no change to BGP, residing as a layer on top of existing BGP implementations. A consensus router logs the output of the BGP policy engine locally, and uses the global consensus algorithm only to determine the most recent consistent BGP state; thus, an AS does not disclose any more information about its preferences than with BGP. We believe the consensus routing design also applies to intradomain routing without the need for a central policy engine.

In summary, our primary contribution is a simple, practical routing protocol that allows general policies and achieves high availability. To this end, we present

1. Consensus routing, a policy routing protocol that systematically separates safety and liveness concerns using two modes: a stable mode to ensure consistency, and a transient mode to optimize availability.
2. Provable guarantees that packets traverse adopted loop-free routes under general policies.
3. Experimental results based on the current Internet graph that show that consensus routing achieves high availability amidst link failures and policy changes.
4. A proof-of-concept prototype of consensus routing based on XORP [17] showing that the proposed design is practical and incurs little processing overhead.

2 A case for consistency

We illustrate several examples where inconsistent forwarding tables cause transient unavailability in inter- and intra-domain routing. These examples are well known, and while some solutions have been proposed to address each, our contribution is a comprehensive but simple solution to the suite of problems. In each case, the unavailability could last several tens of seconds (and sometimes minutes) due to BGP message processing and propagation delays [23]. For an introduction to BGP, refer to [36].

1. *BGP link failures:* Figure 1 shows how link failures cause transient loops in BGP. Bold lines show selected

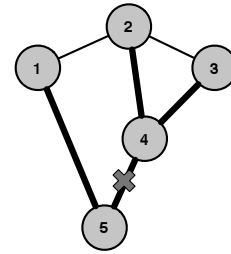


Figure 1: Link failure causing BGP loops at 2 and 3.

paths. If link 4-5 goes down, 4 would immediately send a withdrawal to 2 and 3. However, because both 2 and 3 know of alternate paths 3-4-5 and 2-4-5 respectively, they start to forward traffic to each other causing loops. The MRAI timer prevents 2 and 3 from advertising the new paths even though they have adopted them to forward traffic. The timer is believed necessary to prevent a super-exponential blowup in message overhead, and its recommended value is 30 seconds. Eventually, when the timer expires, both 2 and 3 discover the alternate path to 5 through 1 that existed all along.

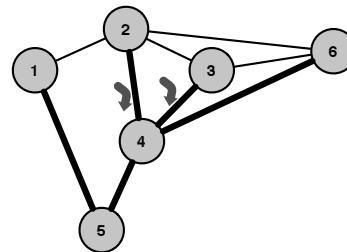


Figure 2: Policy change causing BGP loops at 2 and 3 when 4 withdraws a prefix from 2 and 3 but not 6.

2. *BGP policy change:* Figure 2 shows an example of how policy changes cause routing loops. AS4 may wish to engineer its traffic by withdrawing a prefix from 2 and 3 while continuing to advertise it to 6 for load balancing purposes [34]. (For instance, by diverting traffic to arrive from 6 instead of 2, internal congestion within AS4 might be decreased.) If 2 and 3 each prefer the other over 6, routing loops would result like in Figure 1. A similar situation also occurs if 5 wishes to switch its primary (backup) provider from 4 (1) to 1 (4); in this case, 5 is forced to either withdraw the route advertised (and potentially being used) to 4, or wait for a reliable indicator of when all traffic has completely moved over to the new primary provider 1. Other gadgets involving longer unavailability due to policy changes may be found in [27, 30].

3. *iBGP link recovery:* Figure 3 shows a transient black-hole caused by iBGP inconsistency. Routers A, B, and C belong to AS1 while D belongs to the adjacent AS2. iBGP is a BGP protocol that runs between routers inside an AS (in this case, A, B, and C). All routers route via D

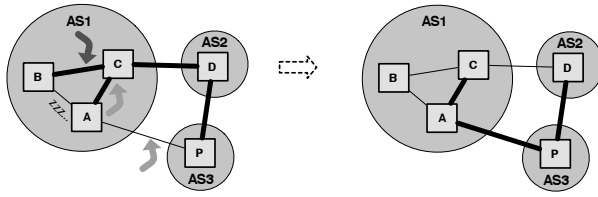


Figure 3: iBGP link recovery causing blackholes.

to the destination P in AS3. Suppose the previously failed link A-P recovers and is preferred by all AS1 routers over the route via AS2. If the AS1 routers all peer with each other, C will withdraw C-D-P from both A and B when it hears from A that A-P is available, but will leave it to A to announce AP to B directly because of the full-mesh design. If A is waiting upon its iBGP timer, B experiences a transient blackhole. The current BGP spec recommends an iBGP timer shorter than interdomain timers, and typical values range from 5-10 seconds. Wang et al. [38] note that such blackholes can cause packet loss for tens of seconds. If AS1 routers use route reflection as opposed to full-mesh, similar consistency problems can cause unavailability [13, 3].

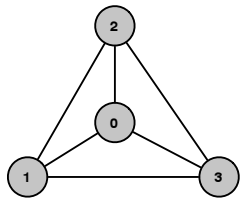


Figure 4: BGP policy cycles causing forwarding loops

4. *BGP policy cycles*: Figure 4 shows the classic “bad gadget” [23, 12] involving cyclic preference dependencies. Each of ASes 1, 2, and 3 prefer to route via its clockwise neighbor over the direct path to AS 0, and does not prefer a path of length 3. The routes will never stabilize because there is no configuration where no AS wants to change its route to AS 0. Furthermore, the system goes through many repeated states involving routing loops causing chronic unavailability.

Summary Some of the specific scenarios illustrated above can be alleviated using known solutions. For example, root cause notification (RCN) [29, 21] can prevent routing loops caused by link failures: in Figure 1, if 4 immediately floods the cause of the withdrawal—the failure of the link 4-5—to 2 and 3, they can prevent the loop. The example in Figure 2 can be addressed if 4 explicitly reveals its intended traffic engineering policy to 2, 3, and 6. However, it appears difficult to extend RCNs to prevent loops or blackholes induced by more sophisticated policy changes, as also noted by [29, 27]. As a result, most ISPs avoid interdomain traffic engineering whenever

possible[1]. The iBGP example shown in Figure 3 can be alleviated using an approach similar to RCP that computes consistent routes in a logically centralized manner. Cyclic policy preferences as in Figure 4 are believed to be rare; Gao and Rexford [7] showed that if ASes restrict their policies to satisfy the “valley-free” and “prefer-customer” properties, such unstable configurations cannot occur. In practice, ASes may have more general policies that do not obey these restrictions [33].

Our goal is to design a single mechanism that can address all of these consistency problems in policy routing. Our solution, consensus routing, is a simple decentralized routing protocol that allows ASes to adopt general routing policies while ensuring high availability.

3 Consensus routing overview

The key insight in consensus routing is to cleanly separate safety and liveness concerns in policy routing. *Safety* means that a router forwards a packet strictly along the path adopted by the upstream routers unless if the adopted route encounters a failed link or router. Note that interdomain routing today does not satisfy this property, e.g., packets may traverse a loopy route even though no router has adopted that route. *Liveness* means that the network 1) reacts quickly to failures or policy changes, and 2) ensures high end-to-end availability defined as the probability that a packet is delivered successfully. By separating safety and liveness concerns, consensus routing achieves high availability while allowing ASes to exercise arbitrary routing policies.

Consensus routing achieves this separation using two simple ideas. First, we run a *distributed coordination* algorithm to ensure that a route is adopted only after all dependent routers have agreed upon a globally consistent view of global state. The adopted routes are *consistent*, i.e., if a router A adopts a route to a destination via another router B, then B adopts the corresponding suffix as its route to the destination. Note that consistency implies loop-freedom. Second, we forward packets using one of two logically distinct modes: 1) a *stable* mode that only uses consistent routes computed using the coordination algorithm, and 2) a *transient* mode that heuristically forwards packets when a stable route is not available.

In BGP, a router processes a received update using its policy engine, adopts the new route in its forwarding table, and forwards the change to its neighbors. In comparison, a consensus router simply logs the new route computed by the policy engine. Periodically, all routers engage in a distributed coordination algorithm that determines the most recent set of *complete* updates, i.e., updates that have been processed by every router whose forwarding behavior depends upon the updates. The coordination is based on classical distributed snapshot [5] and consensus [25] algorithms. The routers use the output of

the coordination to compute a set of *stable forwarding tables* (SFTs) that are guaranteed to be consistent.

Packet forwarding by default occurs in the stable mode using SFTs. When routing changes are caused by policy, the entire system simply transitions from the old set of SFTs to a new set of consistent SFTs without causing routing loops and packet loss. When the stable route is unavailable—because the next-hop router is not accessible due to a failure and the resulting update to recover from failure is incomplete—packet forwarding switches to the transient mode. The router explicitly marks the packet as a *transient packet* and fails over to heuristics such as backup routes, deflections, and detour routes to deliver the packet successfully with high probability.

4 Stable Mode

The distributed coordination proceeds in *epochs* and ensures that, in each epoch, all ASes have a consistent set of SFTs. The k^{th} epoch consists of the following steps, explained in detail in subsequent subsections:

1. *Update log*: Each router processes and logs route updates (without modifying its SFT) until some node(s) calls for the $(k + 1)^{th}$ distributed snapshot.
2. *Distributed snapshot*: The ASes take a distributed snapshot of the system. The snapshot is a globally consistent view of all the updates in the system. Some of them may be complete, and some may still be in progress, e.g., updates that were in transit to a router when the snapshot was taken, or were waiting on local timers to expire before being sent to neighboring routers.
3. *Frontier computation*:
 - (a) *Aggregation*: Each AS sends its snapshot report consisting of received updates, with some of them marked incomplete, to *consolidators*, a set of routers designated to aggregate snapshot reports into a global view. Tier-1 ASes are good candidates for consolidators.
 - (b) *Consensus*: The consolidators execute a consensus algorithm to agree upon the global view, and use the view to compute the set of updates that are globally *incomplete*, i.e., not been processed by all ASes whose routing state would be invalidated by the updates.
 - (c) *Flood*: The consolidators flood the set of incomplete updates I and the set of ASes S that successfully responded to the snapshot, back to all ASes.
4. *SFT computation*: Each AS uses this set of global incomplete updates and its local log of received updates to compute its $(k + 1)^{th}$ SFT, adopting routes carried in the most recent complete update (i.e., not in I) and only involving ASes in S .
5. *View change*:

- (a) *Versioning*: At epoch boundaries, each router maintains both the k^{th} and the $(k + 1)^{th}$ SFT. Each packet is marked with a bit indicating which SFT must be used to forward the packet.

- (b) *Garbage collection*: ASes discard the k^{th} SFT after the $(k + 1)^{th}$ epoch has ended, i.e., when the $(k + 2)^{th}$ snapshot is called for and distributed.

Sections 4.1–4.5 elaborate the above five steps and 4.6 lists safety and liveness guarantees. ASes are assumed failure-prone but not malicious. We discuss the implications of malicious ASes in Section 7. Before that, we briefly comment on the feasibility of the approach.

First, not all ASes need participate in the protocol in order to ensure loop-freeness. A stub AS can not be involved in a loop since no AS transits traffic through it. So, only transit ASes (≈ 3000) participate in the protocol, an order of magnitude reduction compared to the total number of ASes (≈ 25000).

Second, ASes do not send all received updates to the consolidators. Instead, they only send identifiers for updates received in the previous epoch, and the consolidators send back a subset of the identifiers corresponding to updates deemed incomplete. Our evaluation (Section 6) shows that the additional overhead due to consensus routing is a small fraction of BGP's current overhead.

Third, a small number of consolidators (e.g., about ten tier-1 ASes) suffice. The consolidators run a consensus algorithm [25] once every epoch, whose communication overhead is a small fraction of the dissemination overhead above.

4.1 Router State, Triggers, Update Processing

Router State: A consensus router maintains the following state:

1. *Routing Information Base (RIB)*: stores for each prefix the most recent (i) route update received from each neighbor, (ii) locally selected best route, (iii) route advertised to each neighbor; this is identical to BGP's RIB.
2. *History*: stores for each prefix a chronological list of received and selected routes in the RIB. A *received update* is added to the *History* when an update is processed, and a *selected update* is added when the best path to a prefix changes.
3. *Stable Forwarding Table (SFT)*: stores for each prefix the next-hop interfaces corresponding to the stable routes selected for the current and previous epochs.

Triggers: Consensus routers maintain the following invariant: if a router A adopts a new route to a destination, then every router that had received the old route through A has processed the update informing it of the change. Triggers are used to maintain this invariant.

A *trigger* is a globally unique identifier for a set of causally related events propagating through the network.

A trigger is a two-tuple: (*AS number*, *trigger number*). The first field identifies the AS that generated the trigger. The second field is a sequence number that is incremented for each new trigger generated by that AS.

In BGP, each update announces a route and implicitly withdraws the previously announced route. In consensus routing, each update additionally carries a trigger that is associated with the route being implicitly withdrawn and replaced by the route announced in the update. The trigger essentially tracks when the implicit withdrawal is complete, i.e., when all routers that had previously heard of the old route have processed the update. For uniformity, we assume that a BGP withdrawal message is an update announcing a *null* route that withdraws the previously announced route.

To maintain our invariant, we have the following rules for associating triggers with updates. An AS *A* generates a new trigger to be sent along with an update upon 1) a failure of the adjacent link or the next-hop router in *A*'s current route to the destination, 2) an operator-induced local policy change that causes *A* to prefer another route to the destination than the current one, or 3) receiving a route from a neighbor *B* that it prefers over its current route via a different neighbor *C*. Otherwise, when the received update (from *B*) implicitly withdraws *A*'s current route (via *B*) to the destination, *A* simply propagates the trigger associated with the received update.

Update Processing The procedure `PROCESS_UPDATE` presents the pseudocode for processing and propagating updates and their triggers during each epoch. For simplicity, we assume that 1) all updates are for a single prefix, and 2) each AS is a single router; we relax both assumptions in Section 4.7. Upon receiving an update from AS *B* with trigger *t*, AS *A* does as follows:

`PROCESS_UPDATE(B, r, t):`

1. Add the update's trigger *t* to the local set of incomplete triggers I_A .
2. Process the update as in BGP. Let *old* and *new* be the best route to the prefix before and after the update. We define *next_hop* for a route to be the first AS in the route.
3. Add the *received update* (*t*, *r*) to the head of the *History* list. Consider the following cases:
 - (a) *old.next_hop* is not *B*, and *new.next_hop* is not *B*: do nothing since the best route has not changed.
 - (b) *old.next_hop* is not *B*, and *new.next_hop* is *B*: propagate *new* to neighbors with trigger *t'*, where *t'* is a newly generated trigger. Add the *selected update* (*t'*, *new*) to the start of *History*.
 - (c) *old.next_hop* is *B*: propagate *new* to neighbors with unchanged trigger *t*. Add the *selected update* (*t*, *new*) to the start of *History*.
4. Remove *t* from I_A .

An AS marks a trigger as incomplete if it has not yet fully processed an update carrying that trigger. "Process-

ing" an update means both running it through its local policy engine and reliably propagating the resulting update to its neighbors. Thus, if an update is waiting for the MRAI timer at an AS, its associated trigger is marked incomplete. Incomplete triggers at the time of the global snapshot are excluded from the SFT for the next epoch. To ensure consistency of routes, an AS does not adopt a new route until it knows that the trigger associated with the corresponding update is complete.

4.2 Distributed Snapshot

Routers transition from one epoch to another by taking a distributed snapshot of global routing state. The local image corresponding to AS *A* consists of the sequence of triggers \bar{H}_A stored in *A*'s *History*, and the set of incomplete updates \bar{I}_A . An update can be incomplete at an AS when the snapshot is taken because: (i) the update is being processed by the AS (and is therefore in I_A), (ii) the AS might have processed a received update, but the resulting update to a neighboring AS is waiting for the MRAI timer to expire, or (iii) the update is in transit from a neighboring AS.

To initiate a distributed snapshot, an AS saves its local state and sends out a special marker message to each of its neighbors. When an AS *A* receives a marker message for the first time, it executes the following procedure:

SNAPSHOT:

1. Save the sequence of triggers in *History* as \bar{H}_A .
2. Start logging any triggers received on channels other than the one on which the marker was received.
3. Initialize the set of incomplete triggers \bar{I}_A to ϵ . Add the set of triggers in I_A to \bar{I}_A ; these triggers correspond to the updates currently being processed.
4. Scan the outgoing queues for updates waiting on MRAI timers to expire, and add their triggers to \bar{I}_A .
5. Send a marker to all neighbors.
6. Stop logging triggers on a channel upon receiving a marker on that channel.
7. Once the marker has been received on all channels, add logged triggers to \bar{I}_A . These correspond to updates in transit during the snapshot.

The above algorithm is essentially the Chandy-Lamport snapshot algorithm [5] and can be initiated by any AS in the system. A consistent view is obtained even when multiple ASes initiate the snapshot operation concurrently, and we thus require each of the consolidators to initiate the snapshot based on locally maintained timers. The distributed state (\bar{H}_A and \bar{I}_A) across all ASes is aggregated in order to compute a *frontier*, i.e., the most recent complete update at each AS, as described next.

4.3 Frontier computation

The frontier computation consists of three steps:

Aggregation: After the snapshot, each AS A sends to all consolidators the following *snapshot report*:

1. The set of incomplete triggers \bar{I}_A .
2. The saved sequence of triggers \bar{H}_A .

Consensus: Typically, Tier-1 ASes act as replicated consolidators. Replicated consolidators ensure that (i) there is no single point of failure, (ii) no single AS is trusted with the task of consolidating the snapshot, (iii) a consolidator is reachable from every AS with high probability.

The consolidators wait for snapshot reports for a specified period of time. Then, they exchange received snapshot reports in order to propagate reports sent only to a subset of the consolidators to all consolidators. The message exchange does not guarantee that all consolidators have the same set of reports. So, consolidators run a consensus algorithm (such as Paxos [25]), to agree upon the set of ASes, S , that have provided \bar{I}_A and \bar{H}_A . Consolidators propose a value for S by communicating the reports that they have received to a majority of consolidators. The communication can be optimized to avoid the transmitting reports already available at the recipients. The majority then picks one of the proposed set of reports as the consensus value, and this value is propagated to the rest. Paxos is safe, but not live: in no case will the consolidators disagree on the set of ASes S , but if the consolidators fail repeatedly (unlikely if they are Tier-1s), then progress may be delayed.

When the consensus protocol terminates, each consolidator uses the snapshot reports \bar{I}_A and \bar{H}_A of each AS $A \in S$ to compute the set of incomplete triggers I in the network. This set I is computed using the procedure `COMPUTE_INCOMPLETE` which works as follows. A trigger is incomplete if present in an any \bar{I}_A . A trigger is said to *depend* on all triggers that precede it in any \bar{H}_A . This property ensures that any causal dependencies between updates is captured by our system. A trigger t is defined *complete* only if neither t nor any trigger it depends on is incomplete. Therefore, if a trigger is incomplete, then all triggers that follow it in any \bar{H}_A would also be considered incomplete.

`COMPUTE_INCOMPLETE($S, \bar{I}_A[], \bar{H}_A[]$):`

1. Initialize $I = \bigcup_{A \in S} \bar{I}_A$.
2. Do until I reaches a fixed point:
 - (a) For each $t \in I$, for each A do:
 - i. if t occurs in \bar{H}_A , add the first occurrence of t and all subsequent triggers in \bar{H}_A to I .

Flood: The set I of incomplete triggers in the network enables ASes to determine the most recent complete frontier. The consolidators flood the set of incomplete triggers I and the membership set S as computed above to all the ASes. At the end of this flooding phase, every AS uses the same global information about incomplete triggers I and

the set of ASes S to compute the stable forwarding table for the next epoch. Note that a simple optimization here allows us to reduce the size of the flood message, by not sending the complete set I . Since the consolidators know the sequence \bar{H}_A for each AS $A \in S$, they need to only send the first trigger from each \bar{H}_A that is incomplete.

4.4 Building Stable Forwarding Table

After an AS receives the set of incomplete triggers I from the consolidators, it builds a new SFT and readies its state for the next epoch. The procedure for an AS A 's router to build its SFT is as follows:

`BUILD_SFT(I, S):`

1. Copy the current SFT to be its previous SFT.
2. For each destination prefix p :
 - (a) Find the latest *selected update* $u = (t, r)$ in p 's *History* such that t is complete, i.e., neither t nor any preceding trigger is in I .
 - (b) Adopt r as the route to p in the new SFT.
 - (c) Drop all records before u from p 's *History*.

Step 2 above adopts the most recent route update for a prefix such that the trigger associated with it is complete. If any adopted path contains an AS whose snapshot report was excluded by consensus, then the corresponding route is replaced by *null* in the SFT. This ensures that a slow or failed AS is not used to transit traffic in stable mode. Section 5 presents transient mode techniques that improve packet delivery in this case.

4.5 View change

Versioning: The end of `BUILD_SFT` marks the end of the k^{th} epoch and the beginning of the $(k+1)^{th}$ epoch. Since ASes do not have synchronized clocks, different ASes make this transition at slightly different times. To ensure consistent SFTs, ASes maintain and use both the k^{th} and $(k+1)^{th}$ SFT in epoch $k+1$.

Packet forwarding at epoch boundaries proceeds as follows. Once a router has computed the $(k+1)^{th}$ SFT, it starts forwarding data packets using the new routes. Along the way, if a packet reaches a router that has not finished computing the $(k+1)^{th}$ SFT, the router sets a bit in the packet header, and routers forward the packet along the route in the k^{th} SFTs from that point onwards. (A single bit in the header suffices if packet transit times are less than the epoch duration. If not, we could use two bits in the header, and packets older than one epoch are forwarded using transient mode.) Once routers start forwarding a packet on the older route, the packet continues on that route until it reaches the destination. Disallowing the packet to switch back to routes in the $(k+1)^{th}$ SFTs ensures loop-free forwarding.

Garbage collection: ASes discard the k^{th} SFT after the $(k+1)^{th}$ epoch has ended, i.e., when the $(k+2)^{th}$ SFT

has been computed. Discarding older tables ensures that slow or failed ASes do not consume excessive resources at other ASes. In the $(k + 2)^{th}$ epoch, if an AS receives a packet sent using a route from the k^{th} epoch or before, it simply treats the packet as if the corresponding route were *null*, and switches to the transient forwarding mode.

4.6 Safety and Liveness

Consensus routing generates consistent SFTs, i.e., if a router r_1 adopts a route r_1, r_2, \dots, r_k, P to a prefix P , then each intermediate router $r_i, i \leq k$ adopts the route r_i, \dots, r_k, P to that prefix. Equivalently, two routes destined to the same prefix are defined consistent if they share a common suffix starting from the first common router. Note that consistency implies loop freedom. Consensus routing achieves consistency by design; if the downstream AS adopts a recent path, then consensus routing ensures that the withdrawal of the previous path is complete and that no upstream AS is using the old path.

Consensus routing also provides the following liveness property: If an AS selects a sequence of routes $R_1, R_2, \dots, R_i, \dots$ to some prefix, then for each i , it will eventually adopt some route $R_j, j \geq i$, under restricted failure assumptions. Consensus routing achieves this by ensuring that each AS can declare a trigger to be incomplete only for a finite period of time and can send out updates with a given trigger at most once.

These properties are formally stated and proved in the technical report [19].

4.7 Extensions

4.7.1 Multiple routers in an AS

Consensus routing, as presented above, can safely accommodate multiple border routers in an AS. Each border router plays the role that an AS plays above. Consistency safety is preserved as routers only adopt complete updates. However, this naive approach 1) does not scale well as some ASes may have several tens of border routers, 2) does not reflect the administrative unity of policies adopted by these routers.

To address these problems, consensus routing designates one (or more) router(s) in each AS as a local consolidator. The local consolidator collects the snapshot reports from each border router, sends summary reports to the Tier-1 consolidators, and gets back the set of incomplete triggers that it distributes to the border routers.

4.7.2 Prefix Aggregation

Consensus routers maintains *History* on a per-prefix basis, and so the dependencies between different prefixes is not captured. Prefixes are usually independent except when aggregation occurs. Consensus routing can easily be extended to handle prefix aggregation. If an AS aggregates two prefixes p_1 and p_2 into a larger prefix p , then all

subsequent updates received for p_1 and p_2 are also added to p 's *History*. This ensures that an update to p completes only after the corresponding updates to its component prefixes (p_1 and p_2) have completed.

4.7.3 Transactional BGP

Consistency enables ASes to execute a sequence of updates atomically, e.g., an AS may wish to withdraw a prefix from one provider and announce it to another provider without intermittent disconnection or other unintended behavior. Atomic updates can also enable ASes to smoothly revert from BGP wedgies[10].

4.8 Protocol Robustness

We now describe how we deal with some of the problems that might arise in the face of failures:

AS fails to send its snapshot in time: For the epochs to progress smoothly, ASes have to send their local state to the consolidators in a timely fashion. The consolidators accept snapshot states from other ASes for a period of time after they initiate/receive the snapshot message. When that period ends, the consolidators proceed to send each other the set of snapshots received so that all of them operate on the same information. If an AS fails to get its snapshot to any one of the consolidators, because either the AS is slow/misbehaving or all of the messages to consolidators are lost, then that constitutes a severe AS failure. In such cases, the unresponsive AS will not be used for forwarding traffic in the next epoch, especially since its state could be inconsistent with the rest of the network. The exclusion of such ASes is done by having the consolidators also send out the set of ASes S from whom snapshots were received, along with the consolidated list of incomplete triggers. ASes compute their SFTs as described earlier, but if any selected path contains an AS whose snapshot was not available, then that path is replaced by *null* in the SFT. This ensures that the slow/failed AS is not used for transiting traffic in stable mode. In the next section, we present transient mode techniques that improve packet delivery in the face of such failures.

Consolidator fails: It is possible that the routers collecting the local snapshots from ASes may fail. This could mean that all the consolidators don't operate on the same information, especially if some snapshots are received only by the failed consolidator and they have been partially propagated to other consolidators. Our use of the Paxos consensus algorithm helps us ensure that all the consolidators operate on the same information with respect to ASes' local snapshots.

Recovery: An AS that does not reply to a snapshot request within a timely manner will be marked as failed. (Note that an AS is not allowed to reply to a snapshot if it has not completed the SFT computation for the previous

epoch). We re-integrate a failed AS in the same way that BGP restores failed routers. The ‘failed’ AS exchanges paths with its neighbors, just as it would have had it recovered from a real failure in BGP. At the end of the epoch, if the corresponding triggers are complete, then the new SFTs computed can include the routers in the failed AS, at which point the AS is considered re-integrated. The introduction of new routers is identical to recovery of failed ASes.

5 Transient mode

Forwarding switches to the transient mode when a stable route is unavailable at a router. The stable route may be unavailable due to two reasons. First, the next-hop router along the stable route may be unreachable due to a failure. Routers will not arrive at a consistent response to the failure until the next snapshot. Second, the stable route may be null either because a non-null route has not yet propagated to the router, or some router was slow to submit a snapshot report.

Consensus routing enables a common architecture to incorporate several known transient forwarding schemes such as *deflection*, *detour*, and *backup* routes. Today, in both BGP and intradomain routing, transient schemes [35, 41, 21, 24] are believed necessary to maintain high availability in the light of fundamental limits on convergence times. What is lacking, especially in policy routing, is a mechanism that 1) reliably indicates when to switch to the transient mode and back, and 2) allows different transient forwarding schemes to co-exist. Consensus routing provides this mechanism ensuring that a packet strictly traverses adopted routes unless it encounters a failure, at which point it switches to a failover option in accordance with the AS’s policy preferences.

5.1 Transient forwarding schemes

5.1.1 Routing Deflections

When a packet encounters a failed link, the corresponding router “deflects” the packet to a neighboring AS so that it traverses a different route to the destination. The router consults its RIB and identifies a neighboring AS that announced a different valid route to the destination. If the trigger corresponding to that route is complete, then the neighbor has applied the route to its SFT, and packet delivery is assured as long as the alternate route does not encounter other failed links. Multiple link failures can be handled by adding the identity of each failed link to the transient packet’s header [24] before deflecting it.

If no neighboring AS has announced a different valid route to the destination, the router deflects the packet using *backtracking*, i.e., forwarding the packet back along the link on which it arrived. Backtracking may be extended to multiple hops if the previous AS does not have a different valid route in order to increase the likeli-

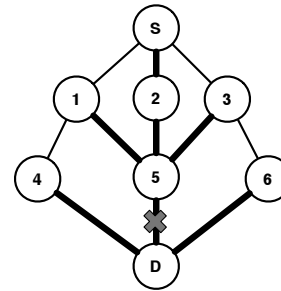


Figure 5: Why deflection packets need more information than BGP. Simple backtracking to provider ASes is not optimal.

hood of successful delivery. Caching information about failed links and pre-computing deflections corresponding to each next-hop further improves lookup times in the forwarding plane.

Unfortunately, backtracking alone is insufficient to guarantee reachability, even if physical routes to the destination exist. Figure 5 gives an example. Bold lines represent chosen best routes to the destination *D*. Each node exports only its best route to its neighbors. For example, node 1 knows two routes to *D*: 1-4-*D* and 1-5-*D*, but it exports to *S* only its best route 1-5-*D*. *S* knows three routes to *D*: *S*-1-5-*D*, *S*-2-5-*D*, and *S*-3-5-*D*. Observe that all three routes go through the same link 5-*D*. Thus, if the link 5-*D* is down, then even backtracking all the way to *S* will not help packet delivery, as none of the nodes *S*, 2, 5 know an alternate route. The next two transient forwarding schemes alleviate this problem.

5.1.2 Detour routing

When a packet encounters a failed link, the corresponding router selects an AS *B* and tunnels transient packets to it. Upon receiving the packet, *B* becomes responsible for forwarding the packet to the destination. If *B* is a Tier-1 AS, there is a high chance of delivering the packet to the destination, since Tier-1 ASes are likely to know of diverse routes to destinations.

This detour approach—having ASes off the standard forwarding path take responsibility for delivering a packet that encounters failures—suggests a potential new business model for ASes where certain ASes advertise themselves as available to deliver packets that encounter failures. Such ASes could either provide the tunneled detouring service for the entire Internet or just for designated prefixes. The tunneling service is similar in spirit to MIRO [40]. Exploring the business model for detour service providers in more detail is beyond the scope of this paper; our focus here is on evaluating the effectiveness of detour routing for transient packet delivery.

5.1.3 Backup Routes

When a packet encounters a failed link, the corresponding router uses a pre-computed backup route to forward the packet. One scheme for pre-computing backup routes to each destination is RBGP [21], which allows ASes to announce backup routes to each other with only slight modifications to BGP. RBGP also shows that choosing the backup route that is most link-disjoint from the primary route protects against *single link failures*—packet delivery is guaranteed as long as a valid route to the destination exists, under certain restrictions on the policies used for selecting and advertising routes. For instance, in the example discussed in Figure 5, node *I* would compute a backup route *I-4-D* and export it to 5, thereby allowing 5 to use the route when its link to *D* fails.

5.2 Implementation Issues

Each of these transient forwarding methods requires some changes to the way packet forwarding works today. Deflection routing requires additional space in the header to store information about the link failures encountered by the packet. Further, backtracking to the previous AS requires packet encapsulation, since the packet will be flowing against the forwarding tables of routers in the current AS. The router that encounters a failed link would have to encapsulate the packet and send it to the ingress router for the AS, which can then send the packet back to the AS it came from. In order to actually backtrack, one would have to keep track of which AS each packet came from, a task that would require additional processing even for stable mode forwarding. To avoid this overhead, backtracking can be approximated by routing the packet towards the source.

Detour routing also requires packet encapsulation, in order to tunnel the packet to the AS that is responsible for delivering those packets that encounter failures. Backup routing techniques like RBGP do not require changes to the packet format, but they require routers to maintain multiple forwarding tables, one for the regular paths and rest for the backup paths.

6 Evaluation

In this section, we explore the effectiveness of consensus routing in maintaining connectivity among ASes, and also analyze the overhead incurred by it. We evaluate consensus routing using 1) extensive simulations on realistic Internet-scale topologies, 2) an implemented XORP [17] prototype, and 3) experiments on PlanetLab. For simulation experiments, we built a custom simulator to compare the behavior of BGP and consensus routing with respect to consistency and availability in the face of failures and policy changes. Our results suggest that consensus routing yields significant availability gains over BGP while ensuring that packets traverse adopted routes, and that

consensus routing adds negligible processing overhead.

6.1 Simulation methodology

We compare standard BGP and consensus routing by simulating routing decisions, protocol control traffic, and link failure on a realistic topology. We use the November 2006 CAIDA AS-level graph [15], gathered from RouteViews BGP tables [16], which includes 23,390 ASes and 46,095 links. CAIDA annotates the link with the inferred business relationship of the linked ASes, namely customer-provider, provider-customer, or peer-peer.

Our simulator simulates route selection and the exchange of route updates between ASes, accounting for MRAI timers. We use standard “valley-free” export policies matching economic incentives, whereby routes through peers and providers are exported only to customers, and customer routes are exported to everyone [7]. We also follow standard route selection criteria, again driven by economic incentives, with customer routes preferred over peer routes, which in turn are preferred over provider routes. If multiple routes fall in the same category, the first tiebreaker is shortest AS path length, and the second is lowest next-hop AS identifier.

We study the routing protocols in three settings: (i) link failures, (ii) traffic engineering accomplished by announcing and withdrawing sub-prefixes, and (iii) traffic engineering accomplished by AS path prepending.

6.2 Link failures

To evaluate the ability of a protocol (BGP or consensus routing with one of the transient mode variants) to cope with failure, we set all routers in our simulator to use that protocol and allow them to reach a stable routing configuration. Then, we conduct a trial for each provider link *L* of each multi-homed stub AS *A*. A multi-homed stub AS is an AS with more than one provider and no customers; our topology includes 9,100 such ASes. We focus on these because the stub AS has a valid physical route to rest of the Internet even if the provider link *L* fails. In each trial, we fail the link *L*, and we record any AS that undergoes a period of disconnectivity to *A* before converging to a new route after the failure. We do not include ASes that the failure permanently disconnects and leaves without any policy-compliant routes. For BGP, an AS is disconnected if it has no route to *A* or if its forwarding path towards *A* includes a loop. For consensus routing, no loops occur, so an AS is disconnected if its SFT route to *A* includes the failed link, and the particular transient mode variant employed in the trial fails to find a valid route to *A*.

6.2.1 Standard BGP

We first simulate BGP and demonstrate that, following failures, disconnectivity is widespread before ASes eventually converge to new policy-compliant routes. Our simulator found convergence times similar to those of Internet

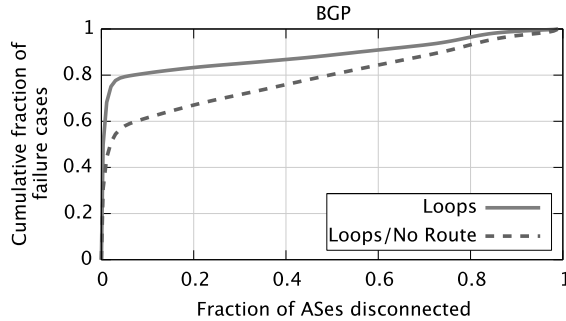


Figure 6: Loops and disconnectivity in BGP following a failure.

measurement studies [23]. Figure 6 shows the prevalence of loops and other disconnectivity (note that a loop is a form of disconnectivity). It shows, for each failure, the fraction of ASes that at some point have no working route to the target AS, before eventually learning a new working route. We see that the failure of just a single link to a multi-homed AS causes widespread disconnectivity, with 13% of failures causing at least half of all ASes to experience routing loops, and 21% of failures causing more than half of all ASes to experience periods of disconnection.

The prevalence of loops argues for the safe application of updates, and the extensive disconnectivity argues for reliable routing in the face of failure.

6.2.2 Consensus routing with transient forwarding

We next evaluate the effectiveness of consensus routing, coupled with various schemes for transient forwarding, at addressing the BGP problems manifest in our simulations. By design, consensus routing ensures that routes adopted at epoch boundaries are loop-free. When adopted routes contain failed links, transient mode routing is invoked. We examine the effectiveness of the following representative schemes for transient forwarding in the face of failures.

Detouring through Tier-1: The router selects the closest Tier-1 AS and detours the packet there. If the Tier-1 does not have a route, it drops the packet.

Deflection by backtracking: The packet is backtracked to the source one hop at a time. If any hop has a failure-free route, it routes the packet to the destination. If the packet reaches the source and the source does not have an alternate route, it drops the packet.

Precomputed backup routes: Routers pre-compute backup routes as in RBGP, which picks the backup route that is most link-disjoint with respect to the primary route. When links fail, policy compliant backup routes are used.

Before invoking any of the above techniques, a router first consults its RIB to ascertain if any neighbor has announced a different valid route R in the most recent complete update. If so, the router safely deflects the transient

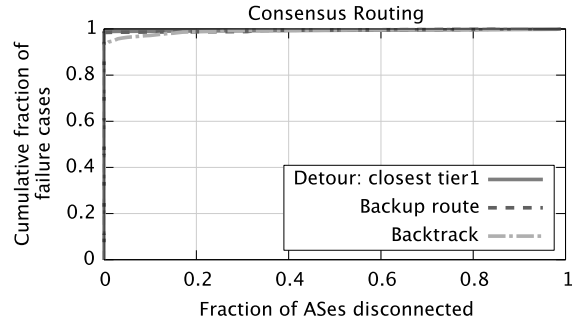


Figure 7: Disconnectivity in consensus routing following a failure.

packet to that neighbor known to have R in its SFT.

Figure 7 shows the effectiveness of consensus routing variants at finding usable routes in the face of failure. Even the simplest forms of transient routing greatly improve connectivity. Backtracking enables continuous connectivity to at least 74% of ASes following 99% of the failure cases. More sophisticated failover techniques further improve connectivity. By detouring to a Tier-1 AS, all ASes maintain complete connectivity following 98.5% of the failures evaluated. Policy compliant backup routes provided similar benefits, with complete connectivity being maintained following 98% of the failures. This experiment suggests that consensus routing can accommodate existing transient forwarding schemes, along with one of the above transient mode variants would provide significantly higher levels of connectivity than BGP, for the failure cases we simulated.

6.3 Traffic engineering by using subprefixes

As described earlier (Figure 2), ASes can engineer traffic by advertising and withdrawing prefixes through a subset of its providers. If ASes advertise their entire range to all providers and selectively advertise their sub-prefixes to control routing, an AS can control its incoming traffic without losing the fault-tolerance benefits of multi-homing. We now evaluate the impact of this form of traffic engineering on route consistency and availability. We evaluate standard BGP and consensus routing using our Internet topology graph, and consider subprefix-based traffic engineering for all multihomed ASes that have three or more providers. There are 3,451 such ASes and 12,890 inter-AS links between these ASes and their providers. In each run, we pick an AS and one of its providers, and withdraw the subprefix from each of the other providers. We then allow the system to converge using BGP and consensus routing, and examine the routes determined by the routing protocols during convergence.

Figure 8 shows the results. For BGP, we see that in more than 55% of the test cases, ASes were disconnected from the destination due to transient loops formed during convergence. Consensus routing transitions from one set of consistent loop-free routes to another, completely

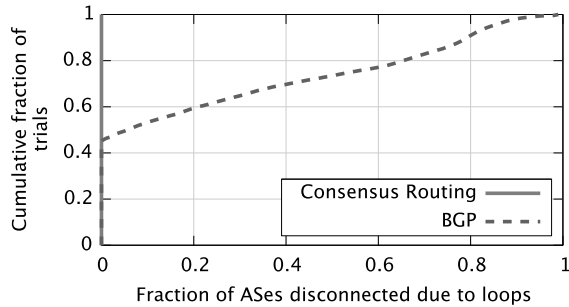


Figure 8: Traffic engineered subprefixes causes loops in BGP.

avoiding transient loops. Note that transient forwarding by itself does not help in this case, as there is no “failure” to trigger the use of transient forwarding or backup routes.

6.4 Traffic engineering by path prepending

We now examine the effect of AS path prepending on the consistency of the routing protocols. Path prepending is another form of traffic engineering wherein a multi-homed AS controls the amount of traffic that flows in through each of its inter-AS links. By prepending itself multiple times on some of its advertised routes, an AS can make the prepended routes less preferable, encouraging upstream ASes to reroute traffic to better paths.

For this experiment, we simulate the routing protocols using the same Internet topology as the previous experiments. In addition, we also model the interaction between BGP and iBGP as this form of traffic engineering is known to cause intra-domain routing loops that in turn lead to routing blackholes [38]. We therefore pick a Tier-1 AS (AT&T for our experiments) and expand its corresponding node in the Internet graph to capture all of its internal topology, while continuing to represent all other ASes as single nodes. We obtained the internal topology for AT&T using the data collected by iPlane [26]. We assume that all the border routers are connected in a full-mesh topology, and model the propagation of routes through the AS according to the iBGP protocol. In this setting, an update is considered incomplete if it is not fully processed by all of AT&T’s iBGP routers. For our evaluation, we consider all the multi-homed stub ASes present in the customer cone of AT&T. In each run of the experiment, we pick one of these stub ASes, which prepends itself three times (our analysis of BGP updates received at the University of Washington showed that most origin ASes prepend their AS number thrice.) on routes to all but one of its providers. We repeat this run for each of its providers.

Figure 9 shows that in more than 20% of the cases, the Tier-1 AS suffers from intra-domain loops due to path prepending by some downstream AS and in many of those instances a significant fraction of other ASes lose connectivity to the target AS.

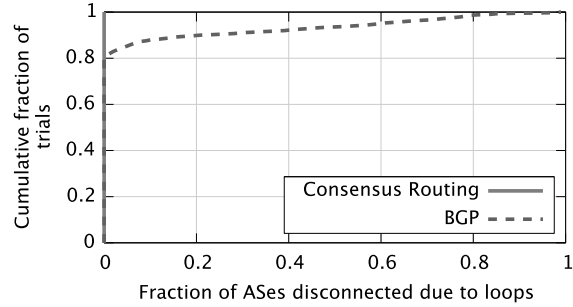


Figure 9: Path prepending causes intra-domain loops in BGP, leading to disconnectivity.

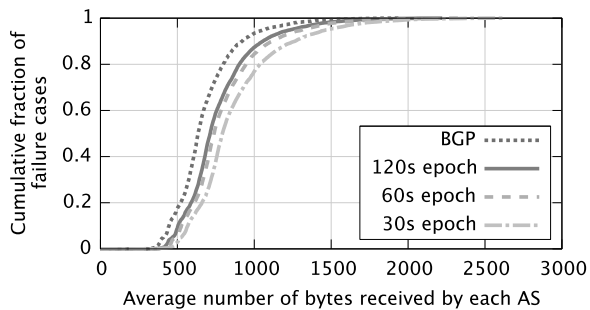


Figure 10: Control traffic required by consensus routing.

6.5 Overhead

The previous sections showed BGP’s inability to maintain continuous connectivity in the face of failure or policy changes, as well as the significantly improved connectivity enabled by consensus routing. We now examine the additional cost incurred by consensus routing compared to BGP.

Volume of control traffic In addition to BGP route update messages, routers running consensus routing must send control traffic to take a distributed snapshot and flood the set of incomplete triggers. Figure 10 shows a CDF of the mean amount of control traffic sent by routers using BGP and consensus routing for our multi-homed stub AS access link failure simulations. The consensus routing overhead was computed for different epoch lengths: 30 seconds, 60 seconds, and 120 seconds.

We observe that the additional overhead introduced by consensus routing is rather negligible. This is because BGP sends updates that are relatively large to all the ASes, whereas the additional traffic sent by consensus routing is mostly sets of triggers, which are smaller; furthermore, this traffic is sent only to transit ASes.

Cost of consensus At the end of each epoch, the consolidators have to reach an agreement on the set of snapshots that will be considered for computing the SFTs. We evaluate the time taken for this process by running Paxos on PlanetLab. In our experiment, we pick 9 random PlanetLab nodes, (each representing a router in a Tier-1 AS), and run Paxos with each of these nodes playing the roles

Number of nodes	Time when first node learns value	Time when last node learns value
9	434 ms	490 ms
18	485 ms	1355 ms
27	590 ms	1723 ms

Table 1: The time taken to run Paxos on 9, 18, and 27 PlanetLab nodes.

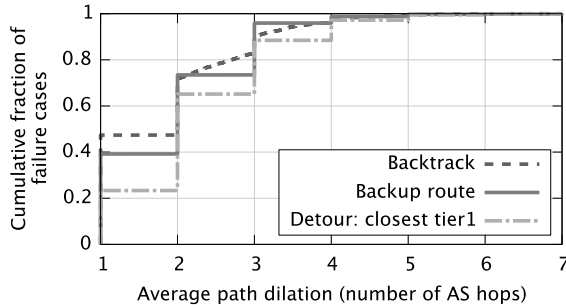


Figure 11: Path dilation incurred by interdomain transient mode options.

of a proposer, an acceptor, and a learner. We measure the time it takes from when a node proposes a value, to when it learns of the chosen value, and the results are available in Table 1. With 9 nodes, all them learn the agreed value in under 450 milliseconds. We then repeat the experiment with 18 and 27 nodes, which are settings that model replication by each consolidator AS to provide fault-tolerance. We observe that the slowest node learns of the chosen value in under 1.4 seconds and 1.8 seconds respectively. We believe that these overheads are acceptable for consensus routing even with epoch durations that are only a few tens of seconds. Further, the time for the consensus phase would be even lesser in a practical deployment, where the loss rates would be significantly lower than what we saw in our experimental PlanetLab setup. We would like to note that any other consensus algorithm, such as Virtual Synchrony [2], can be used instead of Paxos.

Path dilation Figure 11 shows the average amount of path dilation incurred by the various transient mode techniques. Path dilation is the length of the route (in AS hops) traversed by the packet when it encounters failure (from source to failure, then along the detour to the destination), minus the length of the pre-failure route from the source to the destination. It is a measure of how far packets have to be redirected. All the techniques experience a modest amount of dilation, with detouring to a Tier-1 experiencing the most (unsurprisingly).

Response time In BGP, an AS starts using a path as soon as it selects it. However, in consensus routing, an AS has to wait at least till the next epoch boundary before it can change the path it is using. We instrument

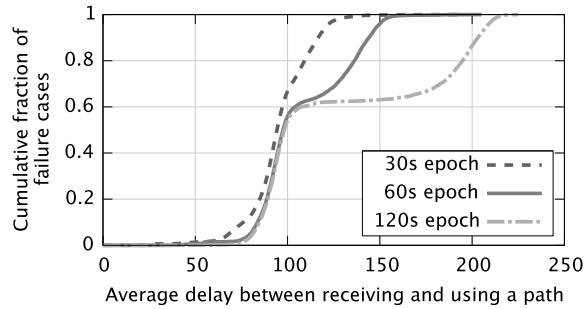


Figure 12: Delay incurred by consensus routing between receiving and using a path.

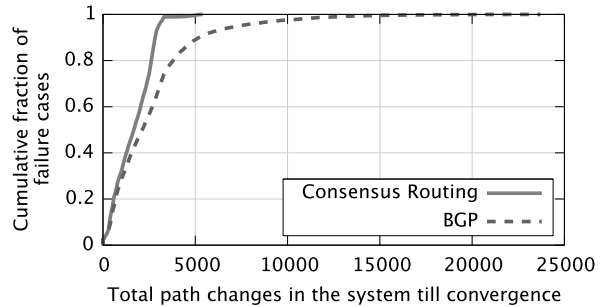


Figure 13: Number of path changes observed in the system.

our multi-homed access link failure simulations to identify the average delay between receiving a path and applying it to the forwarding plane in the case of consensus routing. Figure 12 shows the average delay for different epoch lengths. As expected, shorter epoch lengths allow quicker adoption of new paths. A 30 second epoch results in more than 90% of the paths being adopted in less than two minutes after the path is received. Note that typical BGP path convergence times are of similar duration.

Flux in adopted routes We also measure how many changes are made to routing tables across the entire topology for each routing failure event in our access link failure simulations. In Figure 13, we see that ASes go through several route changes before converging on a stable route in traditional BGP, whereas the convergence takes place through fewer steps in consensus routing. For consensus routing, the number of changes per failure event is typically just one or two per AS, corresponding to the number of epochs the AS takes to stabilize to its eventual route.

Implementation overhead Our XORP prototype was used to measure the implementational complexity, and the update processing overhead at each router. We found that consensus routing added about 8% in update processing overhead, and about 11% additional lines of code to the BGP implementation. More details are available in [19].

7 Security Implications

The separation of packet delivery into two logically distinct modes of packet delivery — stable and transient —

offers other potential benefits with respect to routing security, which we discuss next.

Much prior work has studied the problem of securing BGP against malicious ASes such as SBGP [20], soBGP [39], SPV [18], Listen and Whisper [37]. The primary focus of these efforts is to add cryptographic security to BGP, e.g., to ensure authenticity and integrity of advertised routes; these mechanisms are applicable to consensus routing as well. However, malicious ASes can also use non-cryptographic attacks that exploit protocol dynamics inherent to BGP. For example, a malicious AS can repeatedly announce and withdraw a route causing large shifts in the data plane at ASes much further away [28]. It can also abuse MRAI timers and route flap dampeners intentionally to slow down the responsiveness of other benign ASes to network conditions. MRAI timers are considered necessary to both reduce convergence time and message overhead, and opinions differ on whether or not there is any benefit to further tuning timers [11, 32, 6] even assuming a benign environment.

The stable mode makes routing more deterministic and hence easier to secure. ASes can agree to impose *a priori* specified upper and lower bounds on the interval between snapshots. While we have not implemented this yet, using standard byzantine fault tolerance agreement techniques [4], ASes that are slow or unresponsive either due to benign or malicious reasons can be consistently excluded from the snapshot observed by all ASes provided the network is sufficiently well-connected and the number of byzantine ASes is small. Thus, two good ASes will see the same set of complete updates at the end of an epoch ensuring that their SFTs are consistent. The impact of spurious transient packets can be limited by the following mechanisms - (i) using the failure information only for that packet as opposed to caching it, (ii) limiting the rate at which transient packets can be generated by a single AS as well as in aggregate, (iii) detecting and punishing ASes generating conflicting transient packets. Finally, a malicious AS can still drop all the packets in the forwarding plane after announcing legitimate routes. End-to-end techniques are needed to detect and punish such ASes. In fact, the consistency property—knowing the exact route a packet will take—makes this detection easier.

8 Related Work

Consensus routing, to our knowledge, is the first interdomain routing proposal that allows general routing policies while ensuring high availability. A large body of prior work contributed valuable insights to its design.

Complex dynamics Researchers and practitioners continue to discover complex and unintended effects of BGP dynamics. These include counterintuitive interaction of timers [27]; “wedgies” or unintended stable route configurations that are difficult to revert [10]; a recovering link

causing routing blackholes due to the interaction of intradomain and interdomain routing [38], etc. BGP configuration is considered by some as a black art understood only by a precious few network gurus [36]. This state of affairs calls for making interdomain routing simple to comprehend and manage, which consensus routing achieves by enabling a consistent view of routing state.

Griffin et al. [12] showed that cyclic policy preferences can cause persistent instability. Gao and Rexford [7] showed that by restricting routing policies, stability is guaranteed without global coordination. Consensus routing uses distributed coordination to ensure that ASes transition from one set of consistent (loop-free) routes to another, even under general routing policies. The transient mode ensures high availability throughout.

Consistent Routing: Consistent routing solutions have been previously proposed for intradomain settings. A major advantage of an intradomain settings is that it allows for a logically centralized route computation and information dissemination plane as advocated by RCP [3] or 4D [9]. Luna-Aceves [8] proposed a decentralized approach to consistent route computation for shortest-path distance vector routing, where each node waits for an acknowledgment from the upstream node before adopting the route. The C-BGP proposal by Kushman et al. [22] is a natural extension of this approach to policy routing. Unfortunately, waiting for acknowledgments from upstream ASes means that ASes must rely on timeouts longer than the worst-case BGP convergence delay to ensure consistency; during this time the destination may be unavailable even though a policy-compliant physical route exists. Consensus routing also waits for the withdrawal of the old route before adopting the new route. However, the transient mode ensures high availability even when a stable route is unavailable. The stable mode ensures 1) consistency safety, and 2) progress when more than half the consolidators are up—the epoch length is determined by propagation delays and is not limited by BGP convergence delays. Further, consensus routing not only provides routing consistency, but it can also be extended to support atomic/transactional updates. Transactional updates enable operators to enact multiple traffic engineering operations simultaneously in order to smoothly transition from one set of stable policies to another without causing blackholes and other anomalies.

Triggers in consensus routing are similar to “root cause notification” (RCN) mechanisms proposed previously to weakly track causal relationships between propagating updates to speed convergence [29] or prevent loops [31, 21]. To our knowledge, consensus routing is the first proposal to adapt this mechanism for consistent routing under general policies.

Failure Recovery: The transient mode is similar in spirit to similar proposals for intradomain routing [41, 35, 24].

The proposal by Lakshminarayanan et al. [24] for link-state routing maintains consistent routing tables and encodes information about failed links in packet headers to route around the failure.

R-BGP [21] proposes to use pre-computed backup paths to provide reliable delivery during periods where the network is adapting to failures. Our results on backtracking and finding detours on the fly indicate that packets can be delivered with a high probability even when backup paths are not known, given the current nature of the Internet graph. Furthermore, R-BGP relies on the provider>peer>customer preference for loop-freeness, whereas consensus routing computes consistent routes with general policies.

9 Conclusion

Networking researchers lay great store by the need to improve Internet availability from about two nines (99%) today to four to five nines like other infrastructural services. Although link failure rates can be reasonably expected to improve with time, and consequently drive down periods of transient unavailability to negligible values in intradomain settings, a policy-driven interdomain routing protocol is fundamentally susceptible to long periods of convergence causing transient unavailability. Improving Internet availability to five nines requires us to recognize this fundamental limitation and design around it. Our proposal, consensus routing, is a modest step towards that goal.

Acknowledgments

We gratefully acknowledge Jon Crowcroft for guiding us through the shepherding process. We also would like to thank the anonymous NSDI reviewers for their valuable feedback on the submitted version. This research was partially supported by the National Science Foundation under Grants CNS-0435065 and CNS-0519696.

References

- [1] Personal communication with Aspi Siganporia of Google Inc.
- [2] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, 1987.
- [3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, 2005.
- [4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.
- [5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [6] S. Deshpande and B. Sikdar. On the impact of route processing and MRAI timers on BGP convergence times. *GLOBECOM*, 2004.
- [7] L. Gao and J. Rexford. Stable Internet routing without global coordination. *IEEE/ACM TON*, 9(6):681–692, 2001.
- [8] J. J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. In *SIGCOMM*, 1989.
- [9] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35(5), 2005.
- [10] T. Griffin and G. Huston. BGP wedgies. Network Working Group, RFC 4264, November 2005.
- [11] T. Griffin and B. Premore. An Experimental Analysis of BGP Convergence Time. In *ICNP*, 2001.
- [12] T. G. Griffin, B. F. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM TON*, 10(2), 2002.
- [13] T. G. Griffin and G. Wilfong. On the correctness of iBGP configuration. In *SIGCOMM*, 2002.
- [14] U. Hengartner, S. Moon, R. Mortier, and C. Diot. Detection and analysis of routing loops in packet traces. In *IMW*, 2002.
- [15] http://www.caida.org/data/active/as_relationships/. The CAIDA AS relationships dataset, November 2006.
- [16] <http://www.routeviews.org>. RouteViews.
- [17] <http://www.xorp.org>. XORP: Open source IP router.
- [18] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure path vector routing for securing BGP. *SIGCOMM CCR*, 34(4), 2004.
- [19] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing. Technical Report UW-CSE-08-02-01, University of Washington, 2007.
- [20] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol. *IEEE JSAC*, 18(4):582–592, 2000.
- [21] N. Kushman, S. Kandula, and D. Katabi. R-BGP: Staying Connected in a Connected World. In *NSDI*, 2007.
- [22] N. Kushman, D. Katabi, and J. Wroclawski. A Consistency Management Layer for Inter-Domain Routing. Technical Report MIT-CSAIL-TR-2006-006, MIT, 2006.
- [23] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *SIGCOMM*, 2000.
- [24] K. K. Lakshminarayanan, M. C. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.
- [25] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.
- [26] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *OSDI*, 2006.
- [27] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates internet routing convergence. In *SIGCOMM*, 2002.
- [28] O. Nordström and C. Dovrolis. Beware of BGP attacks. *SIGCOMM CCR*, 34(2), 2004.
- [29] D. Pei, M. Azuma, D. Massey, and L. Zhang. BGP-RCN: Improving BGP convergence through root cause notification. *Computer Networks*, 48(2):175–194, 2005.
- [30] D. Pei, X. Zhao, D. Massey, and L. Zhang. A study of BGP path vector route looping behavior. In *ICDCS*, 2004.
- [31] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. Improving BGP convergence through consistency assertions. In *IEEE INFOCOM*, 2001.
- [32] J. Qiu, R. Hao, and X. Li. The optimal rate-limiting timer of BGP for routing convergence. *IEICE Transactions on Communications*, E88-B(4):1338–1346, September 2004.
- [33] S. Y. Qiu, P. D. McDaniel, and F. Monrose. Toward valley-free inter-domain routing. In *IEEE ICC*, 2007.
- [34] B. Quoitin, S. Uhlig, C. Pelsser, L. Swinnen, and O. Bonaventure. Interdomain traffic engineering with BGP. *IEEE Communications Magazine*, 2003.
- [35] M. Shand and S. Bryant. IP Fast Reroute Framework. IETF Draft, 2007.
- [36] J. W. Stewart. *BGP4: Inter-Domain Routing in the Internet*. Addison-Wesley, 1998.
- [37] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. Katz. Listen and Whisper: Security Mechanisms for BGP. In *NSDI*, 2004.
- [38] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush. A measurement study on the impact of routing events on end-to-end Internet path performance. *SIGCOMM CCR*, 36(4), 2006.
- [39] R. White. Securing BGP through Secure Origin BGP. *Internet Protocol Journal*, 6(3), 2003.
- [40] W. Xu and J. Rexford. MIRO: Multi-path interdomain routing. In *SIGCOMM*, 2006.
- [41] X. Yang and D. Wetherall. Source selectable path diversity via routing deflections. In *SIGCOMM*, 2006.

Passport: Secure and Adoptable Source Authentication

Xin Liu Ang Li Xiaowei Yang
University of California, Irvine
{xinl, angl, xwy}@uci.edu

David Wetherall
Intel Research Seattle & University of Washington
djw@cs.washington.edu

ABSTRACT

We present the design and evaluation of Passport, a system that allows source addresses to be validated within the network. Passport uses efficient, symmetric-key cryptography to place tokens on packets that allow each autonomous system (AS) along the network path to independently verify that a source address is valid. It leverages the routing system to efficiently distribute the symmetric keys used for verification, and is incrementally deployable without upgrading hosts. We have implemented Passport with Click and XORP and evaluated the design via micro-benchmarking, experiments on the Deterlab, security analysis, and adoptability modeling. We find that Passport is plausible for gigabit links, and can mitigate reflector attacks even without separate denial-of-service defenses. Our adoptability modeling shows that Passport provides stronger security and deployment incentives than alternatives such as ingress filtering. This is because the ISPs that adopt it protect their own addresses from being spoofed at each other's networks even when the overall deployment is small.

1. INTRODUCTION

Source authentication in this paper refers to the verification of the source address of a host or network location that originates a packet. The current Internet design trusts each host to place its own IP source address on the packets that it originates, and has at best weak mechanisms to verify that a source address is correct once a packet has entered the network. Because of this, compromised hosts can place incorrect source addresses on packets to impersonate other hosts or obscure the origins of their packets, a practice known as source address spoofing.

Source address spoofing undermines the security and reliability of the Internet in a variety of ways. It enables reflector attacks, in which attackers send initial requests that spoof the address of a victim and trick hosts that reply to send unwanted traffic to the victim. Spoofing obscures the true source of the attack and amplifies it when reply packets are larger than initial requests. Reflector attacks in the early 2006 used DNS servers as unwitting participants to flood the victims with up to 5 Gbps traffic [39].

Spoofing also complicates measures to stop packet floods within the network. Packets' source addresses do not clearly identify the hosts that send them because of spoofing. Consequently, the network cannot automatically fil-

ter packet floods based on source addresses, as attackers may deliberately spoof legitimate hosts' addresses, and filtering would block the legitimate hosts' traffic as well. Similarly, fair queuing based on source addresses cannot effectively limit bandwidth consumed by packet floods. This situation creates a vicious cycle in deploying automated DoS defense mechanisms [17]: the possibility of spoofing leads to the absence of automated defense mechanisms; the absence of such mechanisms obviates the need to attack with spoofed packets, which leads back to the lack of deployment of anti-spoofing mechanisms that makes spoofing possible.

For these reasons, the Internet would benefit from stronger source authentication that makes source addresses trustworthy. Previous work that tackles this problem highlights two extremes in how it can be accomplished. One extreme is ingress filtering [16] in which each AS voluntarily filters spoofed traffic it would originate near the sources, where the legitimate source address ranges are known. This approach is light-weight, but offers limited security benefit and has incentive issues. Specifically, if one network fails to filter spoofed packets, compromised hosts in its network can spoof the addresses of other networks. Up-to-date measurements show that approximately 20% of the prefixes, IP addresses, and ASes on the Internet still allow source address spoofing [7] despite the fact that ingress filtering has been standardized as an Internet Best Current Practice for over seven years. Even when ingress filtering is deployed by all ASes, attackers may still inject spoofed packets if they could compromise routers [42, 50]. This means that ingress filtering provides no guarantee to an AS that it will not have its addresses spoofed at other parts of the network or will not become the victim of reflector attacks, even if the majority of ASes have deployed ingress filtering.

The other extreme is to use strong cryptography-based authentication to verify the source addresses. One example is the approach proposed in [33]. A packet carries a digital signature signed with a source's private key; a router verifies the signature before forwarding the packet. This approach has the adoptability benefit of allowing each AS to independently authenticate the source of a packet without relying on the deployment status or trustworthiness of other parts of the network. As long as an AS has deployed signatures, no attackers can spoof its source address at other ASes where authentication is de-

ployed. However, it requires a per-source public key infrastructure (PKI), and routers need to verify digital signatures at line speed. Both of these requirements are steep and effectively prevent the use of digital signatures at a low level in the protocol stack.

The focus of our work is to understand whether it is possible to achieve the best of both these extremes. Ideally, we would like a source authentication scheme that is as lightweight and incrementally deployable as ingress filtering, yet as robust and beneficial in terms of incentives as digital signatures. To this end, we present the design and evaluation of Passport, a novel network-layer source authentication system. Passport treats an AS as a trust and fate-sharing unit, and authenticates the source of a packet to the granularity of the origin AS. It uses efficient symmetric-key cryptography and checks packets only at administrative boundaries. It leverages the routing system to simply and efficiently manage keys by piggybacking a Diffie-Hellman key exchange on routing advertisements. Together, these properties provide much of the benefits of digital signatures without the corresponding PKI and computational problems.

We implement a prototype of Passport and evaluate its costs and benefits via experiments, security analysis, and adoptability modeling. Our results show that a commodity software PC router can generate or verify packets at up to 2Gbps with an average packet size. We also run experiments on the Deterlab [11] testbed to show how Passport can weaken reflector attacks. We use the adoptability modeling framework presented in [10] to compare the security benefit of Passport with partial deployment with that of other approaches. Our analysis shows that Passport provides stronger security benefit with partial deployment, and hence it is more adoptable than non-cryptography-based approaches such as ingress filtering [16] and route-based filtering [26, 31].

The rest of the paper describes the design, implementation, and evaluation of Passport in greater detail.

2. PROBLEM

2.1 Source Authentication Goals

Our goal is to use source authentication to prevent spoofing under the threat model given in the next subsection. A perfect scheme would check every packet that arrives at each router and verify that the packet carries the source address of the host that injects it into the network; packets with spoofed addresses would be identified precisely and discarded. However, this perfect scheme is unattainable, and we relax the goals of source authentication to permit more realistic designs.

First, we relax the granularity of source authentication. Our design treats an AS as a trust and fate-sharing unit. That is, it only prevents hosts in one AS from spoofing the addresses of other ASes. As each AS is separately

administered, we consider it to be an internal issue for an AS to prevent a malicious host in its network from spoofing the addresses of other hosts in its network. Each AS can use whatever method it prefers to do so. Note that this has the further benefit of allowing source addresses to be authenticated only at the boundaries between ASes, rather than at every router.

Second, we do not verify the freshness of each packet. That is, we do not distinguish between an authentic packet and a replay (a subsequent copy) of the same packet that is injected along the same network path as the authentic packet. This has the downside that packets may be duplicated at any point along the network path and still be considered authentic. While it would be desirable to weed out duplicates, this requires more processing than what we think belongs in the lowest layer of source authentication, e.g., per-source sequence numbers with state kept in the network. Moreover, it is not clear that duplicate packets are as problematic as spoofed packets. Small numbers of duplicates can be detected at end-systems by provisions above the network layer. Large numbers of duplicates could be filtered out in the network in cases where the benefits outweigh the costs. Or even if they are not, large numbers of duplicate packets would likely help in pinpointing the location of duplication.

2.2 Threat Model

The key threat we are concerned with is that attackers can send packets with source addresses that belong to other hosts or routers, yet have those packets pass source authentication checks in the network. We assume that in a realistic Internet environment, both hosts and routers can be compromised, although routers are compromised less often than hosts. This leads us to consider three types of attackers, each with different capabilities.

- **Compromised Host:** This attacker can inject packets into the network with arbitrary source addresses, but cannot eavesdrop the traffic sent by legitimate sources. It is referred to as a Host attacker.
- **Compromised Monitor and Host:** This attacker can eavesdrop traffic (sent by legitimate sources) at its network location and replay that traffic from other, compromised host locations in the network. It is referred to as a Monitor attacker.
- **Compromised Router and Host:** This attacker can eavesdrop traffic (sent by legitimate sources) at its location, and alter or replay that traffic at its location as well as replay it from other, compromised host locations in the network. It is referred to as a Router attacker.

Note that a Router attacker is a strong adversary that can cause greater harm than source authentication failures. We consider such attackers to better understand the

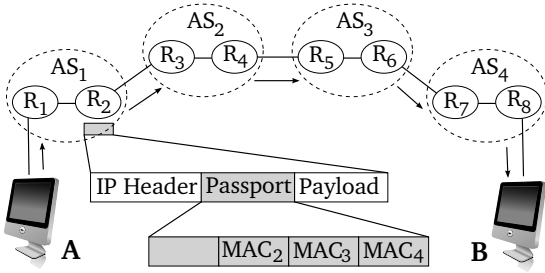


Figure 1: A border router of a source AS (R_2) stamps source authentication information into the Passport header of an outbound packet. A border router of an intermediate or destination AS (R_3 , R_5 , or R_7) verifies this information.

properties of our design. We also study the weaker Monitor attackers to tease apart the properties of Passport.

3. DESIGN

This section describes the baseline design of Passport. For clarity, we ignore the deployment issues in this section and discuss them in later sections.

3.1 Overview

Figure 1 shows how Passport works at a high level. When a packet leaves its source AS, the border router stamps one Message Authentication Code (MAC) for each AS on the path into its Passport header. Each MAC is computed using a secret key shared between the source AS and the AS on the path.

When the packet enters an AS on the path, the border router verifies the corresponding MAC value using the secret key shared with the source AS. A correct MAC can only be produced by the source AS that also knows the key. If the MAC verifies, it is sufficient to show that the packet comes from the source AS indicated by its source address. Otherwise, it is an indication that the source address is spoofed, or there is a temporary routing inconsistency. A packet with an invalid MAC is demoted at an intermediate AS and is discarded at a destination AS (§ 3.4). Routers forward demoted traffic in a separate queue with limited bandwidth (§ 4.3).

Next, we describe how two ASes obtain a shared secret key, and how MACs are computed and verified.

3.2 Obtaining Shared Secret Keys

Passport uses the inter-domain routing system for key distribution. It piggybacks a Diffie-Hellman key exchange [12] on BGP routing advertisements. Each AS_i generates a Diffie-Hellman public/private value pair (b_i, r_i) , and includes the public value b_i in its routing advertisement. This routing advertisement will reach all other ASes so that they can set up a forwarding entry to reach AS_i . Similarly, AS_i will receive routing advertisements from all other ASes. Using the public values included in the routing advertisements, AS_i is able to obtain a shared se-

cret key with every other AS_j using a standard Diffie-Hellman construct: $K(i, j) = b_j^{r_i} \bmod p = b_i^{r_j} \bmod p$, in which p is a system-wide parameter. If an AS originates multiple address prefixes, it only needs to choose one representative prefix to piggyback the key exchange information.

AS_i may receive a routing advertisement originated by AS_j from multiple neighbors. AS_i accepts AS_j 's public value in the routing advertisement from its next-hop neighbor to reach AS_j . This binds the security of the Diffie-Hellman key exchange to routing security, because the public value of AS_j accepted by AS_i is forwarded from the reverse forwarding path from AS_i to AS_j . If the routing system successfully prevents AS_i from forwarding packets via an attacker by rejecting a routing advertisement forwarded from the attacker, then the public value of AS_j accepted by AS_i is not forwarded from an attacker. AS_i is able to compute a correct key shared with AS_j , and verify packets from AS_j using that key. Therefore, as long as routing is secure, Passport is secure.

Passport gains additional benefits from distributing the shared secret keys within the routing system. First, it can bootstrap the key distribution. Key distribution is a seemingly chicken-and-egg problem: keys are needed for packet forwarding, but ASes need to send packets to negotiate keys. As routing packets are exchanged before forwarding state is set up, routing has its own authentication mechanisms to accept routing messages without requiring Passport headers, i.e. routers only accept routing messages from known peers. Second, it gains efficiency. Each AS only needs to send one routing advertisement to establish a shared secret key with every other AS. Lastly, its key distribution channel can be made highly resilient to DoS flooding attacks, because the key distribution information enjoys the same forwarding priority as routing messages. If routers forward routing messages with highest priority, Passport's key distribution information is also forwarded with highest priority.

3.3 Stamping

Passport uses efficient symmetric key MACs as the inter-AS authentication information. A border router of an AS stamps a MAC for each AS on the path to the destination when it receives an outbound packet from an internal interface. Each MAC is computed using the key it shares with the AS on the path. The MAC computed for the destination AS covers the source address, the destination address, the IP identifier (IP ID), the packet length field of the IP header, and the first 8 bytes of packet payload. For instance, in Figure 1, when a packet from host A to host B leaves AS_1 , the border router R_2 of AS_1 computes MAC_4 for the destination AS_4 as $MAC_{K(AS_1, AS_4)}(src, dst, len, IPID, payload[0, 7])$. The MAC computed for an intermediate AS also includes the previous AS number. For instance, R_2 computes MAC_3 as $MAC_{K(AS_1, AS_3)}$

(*src, dst, len, IPID, payload*[0, 7], *AS*₂). A router can obtain the AS path information from BGP.

A MAC computation covers the source address field to prevent spoofing. It covers the other fields to detect packets that are sniffed on one path but injected at other network locations. We discuss more on how Passport prevents this type of sniffing and replaying attack in § 7.

Figure 2 shows a Passport header format used in our implementation. A destination MAC is 64-bit long. Each intermediate MAC is 32-bit long if there are more than one intermediate hop to save header space; otherwise it is 64-bit long.

A border router only stamps a Passport header for a packet with a valid source address that is within its own address space, and discards the packet otherwise. This step is similar to egress filtering [16], but it is only an optimization. Passport prevents address spoofing even without it. This is because if a router stamps MACs for a source address outside its address space, the MACs will not verify at downstream ASes (as we will see next), wasting an AS's processing power.

3.4 Verification

When an AS receives a packet from an external interface, it verifies the Passport header using the key it shares with the source AS. The verifying AS uses the source address of the packet to look up the source AS, obtains the shared key, and recomputes the MAC using the shared key and the same packet fields as used by the source AS. An AS can obtain the mapping between a source address and the corresponding source AS from BGP using the AS_PATH path attribute.

If the source address is not spoofed, a router is able to locate the correct key. The re-computed MAC will match the one in the Passport header. This verifies the source AS of the packet. A router erases the MAC value in a packet after verification to prevent offline cryptanalysis.

If the MAC does not verify, a destination AS discards the packet, because the source address must be spoofed. An intermediate AS demotes the packet, because a MAC mismatch may be caused by either address spoofing or temporary routing inconsistency.

If a packet is demoted, a demotion bit in its Passport header is set, and its IP header is also marked with demotion information to convey the demotion status to legacy ASes (§ 4.3). Intermediate ASes forward demoted traffic in a separate queue with limited bandwidth without further verification. A destination AS still verifies a demoted packet, and discards the packet if the MAC is incorrect. If the MAC is valid, the packet is forwarded to its destination host with the demotion mark. End systems may use these marks to mitigate reflector attacks (§ 7.2). We discuss the trade-off between demote and discard in § 9.1.

An intermediate AS discards packets received from an external interface that spoof its own addresses. This step

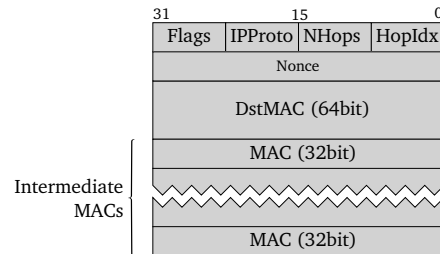


Figure 2: Passport header format.

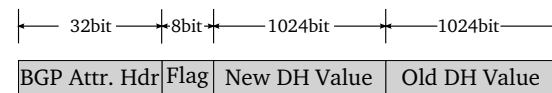


Figure 3: Diffie-Hellman key exchange information is encapsulated in a BGP AS path attribute.

precedes Passport header verification, as it does not require verifying a Passport header.

A router in an AS may receive a packet with a Passport header from an internal interface. If the internal interface is a host-to-router interface, e.g. the interface between host A and a router R_1 in Figure 1, the router discards the packet, as a host can not generate a valid Passport header. If it is a router-to-router interface, it may assume that the packet has been verified by a border router in its AS and forward the packet without further verification.

3.5 Re-Keying

An AS may establish new shared keys with other ASes when its old keys have been used for a while, e.g. on the order of a few days or a few weeks, to improve security. To exchange new keys, AS_i sends a routing update with a new Diffie-Hellman public value, and other ASes compute new keys shared with the AS using this value.

During the re-keying process, different ASes may use different keys to generate the MACs for AS_i , as the routing advertisement of AS_i will arrive at different ASes asynchronously. To identify different keys, an AS uses an alternating parity bit to mark consecutive Diffie-Hellman public values. When AS_j generates a MAC for AS_i , it uses the highest-bit in the MAC field to indicate the parity of AS_i 's public value, and one bit in the Flags field of a Passport header to indicate the parity of its own public value. These two bits uniquely identify a shared key even when both ASes re-key simultaneously.

Figure 3 shows the key exchange information piggybacked in a BGP advertisement. Each advertisement carries both the new and old Diffie-Hellman public values in case a remote router crashes when an AS re-keys. The parity of the new value is indicated in the flag field.

3.6 Key Management and Storage

Diffie-Hellman public values or the shared secret keys are stored with routing information. If a router reboots

and loses those values, it may obtain them from other routers in the same AS, similar to a BGP table transfer after a router reboots.

An AS may configure one router or a route reflector [5] as its key generator. This router is in charge of generating Diffie-Hellman value pairs and initiating re-keying. Other routers learn the Diffie-Hellman private value from the key generator via iBGP.

4. DEPLOYMENT

This section describes how to deploy Passport in the presence of various legacy elements in the Internet. We discuss the high-level issues here. More details such as MTU discovery are discussed in § 9.

4.1 Inter-Operate with Legacy ASes

ASes that adopt Passport may use the optional and transitive path attributes of BGP to distribute their Diffie-Hellman public values as shown in Figure 3. Legacy ASes do not process the optional and transitive path attributes, but will include them in the routing advertisements they propagate to their neighbors [36]. Therefore, two upgraded ASes can perform a Diffie-Hellman key exchange when there are legacy ASes between them.

Passport header is inserted as a shim layer between IP and an upper layer protocol. A source AS stamps MAC values for the upgraded ASes on the path, and legacy ASes do not process the Passport header.

4.2 Bump in the Wire

Passport can be deployed as *bump in the wire* without upgrading hosts. When both a source AS and a destination AS have upgraded, the border router at the source AS inserts a Passport header to a packet, and the border router at a destination AS strips off the header.

If a destination AS has not deployed Passport, an upgraded source AS can still use Passport headers so that other upgraded ASes on the path can verify its packets. In this case, the last upgraded AS on the path strips off the Passport header. However, if there is a temporary routing inconsistency, a Passport header may not be stripped off when the packet reaches its destination. A legacy host in the destination AS may receive a packet with an unknown Passport header and discard it.

To solve this problem, Passport uses IP encapsulation. A source AS encapsulates the original IP packet using an outer IP header. The outer IP header uses the same source address as the inner header, and uses the last upgraded AS on the path as the destination address. This address could be a special well-known anycast address of the last upgraded AS. The source AS inserts a Passport header between this outer IP header and the inner IP header. In this encapsulation mode, a MAC in a Passport header covers the source address, both destination addresses in the inner and outer header, the original 8-byte payload, and the IP

ID and length fields of the outer header.

When the destination AS in the outer IP header decapsulates a packet, it checks whether the incoming AS is a neighbor to which it announces the destination prefix in the inner header. If not, it discards the packet to prevent a source AS from violating its transit policy.

Modern routers already support encapsulation at line speed because of other needs such as VPNs and IPv4-IPv6 transition. After hosts are upgraded, they can process packets with Passport headers, reducing the need for encapsulation.

4.3 Handling Legacy Traffic

In upgraded ASes, legacy and demoted traffic are both unverified traffic, and are treated with the same priority. An upgraded AS may use strict priority queuing to favor verified traffic over unverified traffic, and incent legacy ASes to upgrade. Alternatively, it may use two weighted queues to handle verified and unverified traffic, allocating limited bandwidth to unverified traffic. It may set the queue weights according to the ratio of traffic from upgraded ASes and legacy ones to avoid penalizing legacy traffic when there is no spoofing attack.

An upgraded AS will discard or demote legacy traffic if it detects or suspects that the traffic spoofs other upgraded ASes' addresses as follows:

1. If a legacy packet's source AS and destination AS have both deployed Passport, discards the packet, as it must be spoofed.
2. If a legacy packet's source AS has deployed Passport but the destination AS has not, demotes the packet. This is because a source AS will insert a Passport header for a legacy destination if there is any upgraded AS on the path. A legacy packet from an upgraded AS is likely to be spoofed, except during temporary routing inconsistency.

A legacy AS may receive two types of legacy traffic: regular and demoted by other upgraded ASes. The AS should treat demoted traffic with lower priority, because it is likely to be spoofed.

Upgraded ASes convey demotion information to legacy ASes via the IP header, such as using the Differentiated Services Code Point (DSCP) [29]. A legacy AS needs to make a configuration change to honor demotion in order to take advantage of Passport. We believe this configuration change can be made at most legacy ASes, because DiffServ is already well supported by commercial routers today, and this change does not require software or hardware upgrade. Besides, if a legacy AS encounters congestion in its network, it has to discard some packets. It's advantageous for the AS to honor a demotion mark and discard packets that are likely to be spoofed in favor of regular packets.

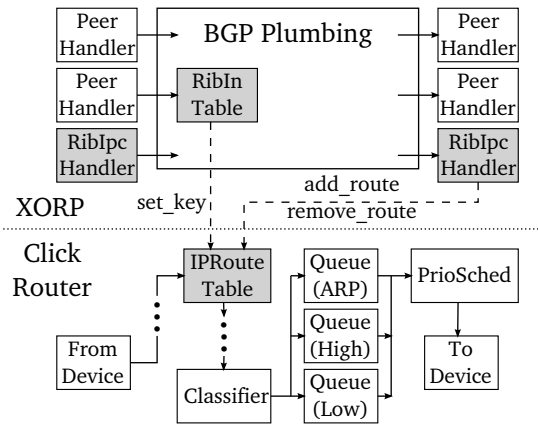


Figure 4: How Passport is implemented using Click and XORP.
The shaded boxes are the main modules we modify.

5. IMPLEMENTATION

This section describes a prototype implementation of Passport. We implement most of the features of Passport using Click [22] and XORP [18]. Figure 4 shows the structure of the implementation. The shaded boxes are modules we modify. We modify XORP to piggyback the Diffie-Hellman key exchange protocol in BGP, and modify components in Click to support Passport header stamping and verification. We also modify XORP to communicate with Click using the `/click` file system.

We add an optional and transitive AS path attribute `DH_KEY` to XORP's BGP modules. This attribute encapsulates Diffie-Hellman public values as described in Figure 3. It is inserted into the BGP prefix advertisements in the module `RibIpCHandler` and extracted from the prefix advertisements in the module `RibInTable`. `RibInTable` is also modified such that whenever new Diffie-Hellman public values are received, the corresponding shared secret key is generated and sent to Click using the `set_key` interface in Figure 4. We also modify `RibIpCHandler` to update Click's routing table with Passport related information using the `add_route` and `remove_route` interfaces in Figure 4. Passport related information includes AS paths and prefix to origin AS mapping. In our current implementation, a new Diffie-Hellman public-private value pair is generated at XORP's startup time. This part needs to be extended to support periodic re-keying, and private key distribution using iBGP.

We modify the `IPRouteTable` element in `Click` such that it receives shared secret keys from `XORP` and calls `generate_ppt()` or `verify_ppt()` in its `push()` method to stamp or verify `Passport` headers. We use `Click`'s priority scheduler to handle normal and demoted traffic as shown in Figure 4. We use priority queuing instead of weighted fair queuing to emphasize the benefit of deploying `Passport`. The `ARP` queue ensures that link-local `ARP` packets have highest forwarding priority.

Our implementation uses UMAC because of its supe-

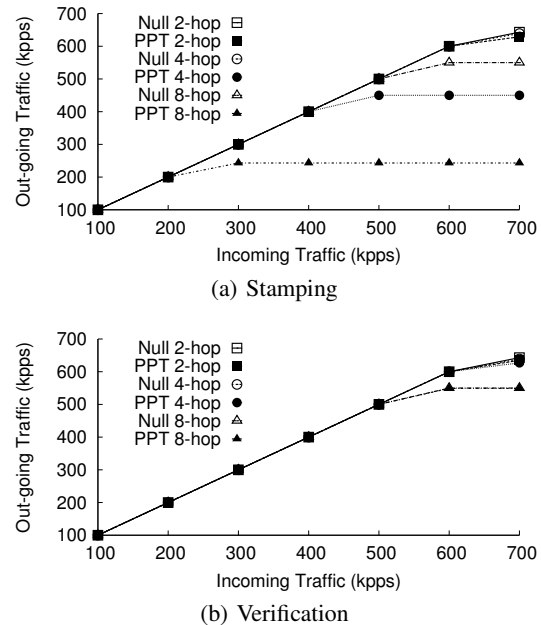


Figure 5: The throughput of Passport header stamping and verification for various AS hops with minimum sized packets (40 byte TCP/IP headers plus a Passport header). The Click null forwarding throughput for packets with the same size are also shown for comparison.

rior speed. UMAC takes a nonce as input. We stamp a random number into the 32-bit nonce field of a Passport header and use it together with the 16-bit IP ID to generate a 48-bit nonce for UMAC computation. In the worst case when IP ID field never changes, the nonce space is 2^{32} , and a nonce may be reused before an AS re-keys. For this reason, we did not use UMAC in [23] because it is vulnerable to a single nonce reuse. Instead, we use the construction in [8] combined with UHASH [23] and AES. This construct is provably robust to occasional nonce reuse. (Due to the lack of space, we cannot include the proof, but we have confirmed our proof with the UMAC inventor.)

6. PERFORMANCE EVALUATION

6.1 Passport Header Processing Overhead

We use three PCs in our laboratory to measure the computational overhead of Passport header stamping and verification. One PC is used as a router, connecting a packet generator PC and a sink PC. The router has an AMD Opteron 285 Dual Core 2.6GHz CPU, 2GB memory, and an Intel PRO/1000 GT Quad Port Server Adapter. The packet generator and sink are equipped with Pentium-D 3.4GHz CPU, 2GB memory and Intel PRO/1000 PT Server Adapter. We measure both the throughput and CPU cycles of Passport stamping and verification.

To measure the throughput of Passport header stamp-

	Operation	Time		
		2-hop	4-hop	8-hop
Per Packet	Passport Stamping	655 ns	1493 ns	3190 ns
	Passport Verification	578 ns	618 ns	631 ns
Re-key	DH value pair (1024-bit)	5.64 ms		
	Symmetric key (128-bit)	5.64 ms		

Figure 6: Micro-benchmark of various Passport operations.
Time is converted from CPU cycles.

	Security	Sig. Size	Signing	Verification
RSA-512	60-bit	64 bytes	512 us	40 us
RSA-1024	72-bit	128 bytes	2214 us	102 us
DSA-512	65-bit	40 bytes	368 us	443 us
ECDSA-160	78-bit	40 bytes	300 us	1400 us

Figure 7: Equivalent MAC security level, signature size and computational costs of well-known public key schemes.

ing, we let the packet generator send minimum sized packets (40 bytes TCP/IP headers) at various input rates. Our experiments assume legacy hosts, and the router PC inserts Passport headers into the packets. The sink measures the output rate. We vary the number of AS hops for each experiment, because a router needs to stamp a MAC for each AS on the path. Note that N AS hops implies $N - 1$ MAC computations. Ten million packets are sent for each experiment.

Similarly, to measure the throughput of Passport header verification, we let the packet generator send minimum sized packets with preset Passport headers at various rates with various AS hops, use the router PC to verify Passport headers, and measure the output rates at the sink.

Figure 5 shows the Passport header stamping and verification throughput, together with Click null forwarding throughput for packets of the same size. The Passport header verification throughput matches well with Click’s null forwarding throughput, as it only involves one MAC computation. The verification throughput varies from 636 kpps to 549 kpps for Passport headers of two to eight AS hops. The slight decrease is primarily due to the increase of packet length, not by the MAC computation.

The Passport header stamping throughput decreases as the AS path length increases. For Passport headers with two to eight AS hops, the throughput varies from 628 kpps to 243 kpps. If we assume that the average packet size is 400 bytes [1], the PC router can stamp Passport headers for average sized packets with two to eight AS hops at 2 Gbps to 0.9 Gbps. As the mode and mean of the AS path length are between 3 and 4 [13], we expect that the stamping throughput for real Internet traffic exceeds 0.9 Gbps. We also note that an AS only needs to stamp Passport headers for traffic originated within its network, and not for transit traffic. 1~2 Gbps stamping throughput might be sufficient for most ASes.

We measure the CPU cycles for Passport header stamping and verification using the *get_cycles()* Linux kernel

function. Figure 6 shows the result, with CPU cycles converted to time. The per-hop increment of a Passport header stamping time is about 420 ns.

Passport-enabled routers also exchange Diffie-Hellman keys on the routing plane. The cryptographic operations include generating Diffie-Hellman public-private value pairs and computing shared secret keys from Diffie-Hellman values. Both Diffie-Hellman value pairs and shared secret keys are generated using exponentiation. We tested the overhead to generate a Diffie-Hellman value pair and to compute a shared secret key on the router PC. As shown in Figure 6, each public key operation takes 5.64 ms.

The public key operations are expensive, but are unlikely to become a performance bottleneck, because an AS only performs public key operations when it re-keys. Re-keying should happen infrequently such as no more than once per week. When an AS itself re-keys, it needs to generate a shared secret key with all other ASes. There are less than 30K ASes seen in BGP routing tables according to data from RouteViews [37]. It takes less than three minutes to generate all shared keys on the router PC. If another AS re-keys, an AS only needs to generate one shared secret key when it receives a new Diffie-Hellman public value from that AS. If we assume all ASes re-key randomly during a period of seven days, then on average, an AS may receive less than three new Diffie-Hellman public values per minute, and spend 17 ms to compute the shared secret keys.

6.2 Memory Overhead

Passport maintains per-AS key information. A router keeps a shared secret key per AS for Passport header stamping and verification. A MAC computation typically requires the initialization of a key context. It is desirable that a router initializes and stores the key context for fast processing. The size of a key context depends on the specific MAC. Our implementation uses a UMAC key context that consumes 388 bytes. It requires less than 12MB memory to store the shared keys and their key contexts.

When an AS re-keys, another AS may need two different keys for Passport stamping and verification: one generated with the old Diffie-Hellman public value, and the other with the new value. This requires additional memory. As shown above, the average re-keying rate of all ASes is less than 3 per minute. If we assume it takes at most an hour for BGP to converge, then the number of simultaneously re-keying ASes is around 180, adding an additional 70KB memory overhead.

6.3 Header Overhead

As shown in Figure 2 and described in § 3.3, a Passport header has a 16-byte fixed header overhead, and four bytes per additional AS hop if the path length exceeds 4 hops, or an additional 8-byte overhead if the path length is 3 or 4. If we assume an average AS path length is four,

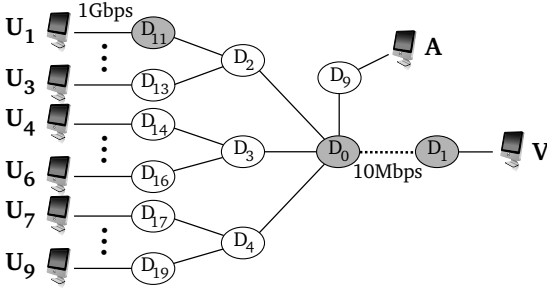


Figure 8: Topology of Deterlab experiments.

the average header overhead is $8 + 8 + 8 = 24$ bytes.

Passport's header and processing overhead is inherent to cryptography-based security mechanisms. For comparison, we list the header and processing overhead of well-known public key signatures that provide a similar level of security as one 64-bit MAC in Figure 7. The tests are done using the *openssl* speed test on the router PC, and the security levels are estimated according to [25].

6.4 Deterlab Experiments

We run experiments on a Deterlab [11] testbed to test the correctness of our implementation. We emulate a scenario in which malicious Host attackers in legacy ASes spoof the source address of a victim in an upgraded AS to launch reflector attacks.

The experiment topology is shown in Figure 8. Each circle represents one AS. Only the shaded ASes deploy Passport. We configure each AS to have only one router D_i . The bottleneck link is between D_0 and D_1 . The victim host V is connected to D_1 , the attacker host A is connected to D_9 , and D_{11} to D_{19} each has one host connected to it. We wish to use a topology with more hosts, but we were limited by the number of PCs we could hold on the Deterlab.

In our experiments, hosts U_1 to U_9 each run a reflector application that replies to incoming UDP packets with an amplification ratio of 40. The attacker spoofs the victim's address and sends UDP packets to all reflectors $U_1 \sim U_9$ uniformly. Each UDP packet sent by the attacker has 32 bytes payload. These parameters are set to emulate a DNS reflector attack [39].

After the attack is started, hosts U_1 to U_9 also each send 100 files to the victim using TCP. These TCP transfers are used to measure how the reflector attack affects the network performance seen by an end system. The size of each file is 20KB, and a file transfer aborts if it cannot finish in 10 seconds. We vary the attacker's sending rate from 1% to 20% of the bottleneck link bandwidth and measure the file transfer time.

The results are shown in Figure 9. Hosts U_2 to U_9 are in non-upgraded ASes. Their file transfer traffic is legacy traffic and competes for bandwidth with the legacy reflector traffic at D_0 . When the reflector traffic congests

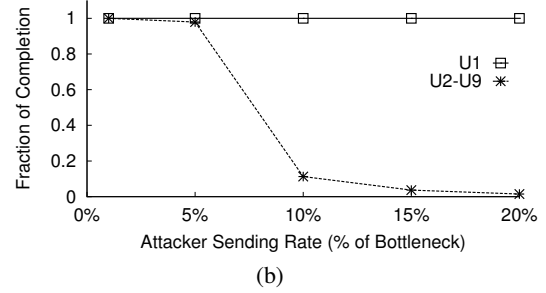
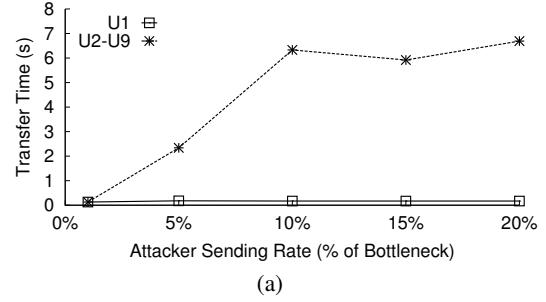


Figure 9: The average file transfer times and fractions of completion for hosts U_1 to U_9 when the attacker A spoofs V 's address to launch a reflector attack. U_1 is in an upgraded AS D_{11} , while U_2 to U_9 are in non-upgraded ASes D_{12} to D_{19} .

the bottleneck link, their file transfer traffic suffers from congestion. The file transfer traffic from U_1 carries Passport headers and only competes for bandwidth with verified Passport traffic at D_0 . Therefore, U_1 is not affected by the reflector traffic and can finish all the file transfers quickly. Note that the reflector application on U_1 will not receive attack traffic and therefore will not send out reflector traffic to compete with U_1 's file transfer traffic at the bottleneck link, because when attack packets to U_1 reach D_0 , D_0 will discard them as they do not include Passport headers but both their source and destination addresses belong to upgraded ASes (See § 4.3).

7. SECURITY

7.1 Spoofing Prevention

Host Attacker: To spoof source addresses, a Host attacker may try to break the MAC that is the heart of Passport. But our design uses a standard MAC scheme with 128-bit keys, which is computationally infeasible to break. The attacker might instead try to guess a valid Passport header by sending packets. Since a Passport header has at least a 63-bit destination MAC value (modulo the parity bit of a Diffie-Hellman public value), an attacker would expect to send at least 2^{62} packets to guess one correct, but the time it takes to send those packets exceeds the period for which the Passport header will be valid. The long-term key distributed via BGP will have advanced in the interim. This attack would also signal a clear anomaly

via a large number of invalid Passport headers.

An attacker may try to guess an intermediate 31-bit MAC value by sending TTL expired packets, and use the echoed IP header and Passport header to observe whether a guess is demoted. But this again is infeasible as ICMP messages are always rate limited.

Monitor Attacker: An eavesdropper may observe valid Passport headers but cannot freely transfer them to another path because a Passport header is bound to one AS path, i.e., its MACs will be found invalid if sent via another AS path. Thus, packets transferred to other paths will be demoted on their paths to destinations, and can not compete for bandwidth with packets with valid MACs.

Router Attacker: Packets duplicated by routers on the forwarding path of a source may reach a destination without being demoted. However, in this case, routing is compromised, and Passport's security is bound to routing security. An AS spoofed by a router on its forwarding path should choose a different path.

If an attacker compromises an AS, it may use the AS's keys at other, unprotected locations in the network to forge a Passport header that spoofs the AS's addresses. But this only implicates the compromised AS, not other parts of the network. Similarly, a compromised router can spoof packets from other addresses in its AS, but this again implicates the AS and may adversely affect its traffic.

Importantly, even if an AS is compromised, it cannot forge a Passport header from another AS. This is because it does not have the secret keys of the other AS. Even colluding ASes can only forge a valid Passport header that shows a packet comes from themselves. They cannot forge a Passport header as if the packet were from an AS outside the colluding set.

We also note that although two ASes share a secret key, they can not use this key to spoof each other's addresses at other ASes, because other ASes use separate keys shared with these two ASes respectively to validate their addresses.

The security of Passport is bound to routing security. We rely on routing to distribute the correct public Diffie-Hellman public values across ASes. By the Diffie-Hellman construction, we do not rely on routers to keep the public values secret from attackers, since it is computationally infeasible to find the long-term pair-wise keys given only the public values. However, if an attacker can successfully hijack a prefix announcement and replace the Diffie-Hellman public value, it can both receive packets for the specific prefix, and send packets as if they were originated from that prefix.

7.2 Reflector Mitigation

A Monitor Attacker may attempt to launch reflector attacks by sniffing packets sent by a victim, and injecting duplicate copies of them from other network locations. Those replayed packets will be demoted because Pass-

port is bound to an AS path. An upgraded host that understands the demotion mark in an IP header should echo back the demotion mark in its reply packets to avoid becoming a reflector.

Unfortunately, if a host is not upgraded to echo back a demotion mark, it may respond to replayed packets without demoting the reply packets, thereby becoming a reflector. Passport alleviates this problem by including the destination address, IP ID, IP length field of an IP header, and 8 bytes of payload in the MAC computation. The 8-byte payload covers the TCP sequence number and UDP checksum. A replayed packet must have the same source address, destination address, IP ID, IP length, and TCP sequence number or UDP checksum as the sniffed packet. These fields can help end systems detect and discard replayed packets. For instance, an upper-layer application or a transport protocol such as TCP may detect a duplicate packet, or a sequence number or checksum mismatch, thereby discarding the packet.

We believe that in practice, these mechanisms can effectively mitigate reflector attacks launched using replayed packets even if a host is not upgraded to echo back a demotion mark. However, if this type of attack becomes a serious concern, Passport can be extended to detect and discard packets with duplicate Passport headers in the network at a higher cost, using a combination of sequence numbers and bloom filters as described in an early version of this work [27].

In the case of a Router Attacker, packets can be duplicated without being demoted. As described above, an AS should avoid forwarding packets via a Router Attacker.

7.3 Security with Partial Deployment

In the incremental deployment phase, Passport prevents the addresses of upgraded ASes from being spoofed at other upgraded destination ASes by Host attackers. If a destination AS is not upgraded, then as long as there is an upgraded AS on the path, packets that spoof the upgraded ASes' addresses will be demoted.

Similarly, a packet replayed on one path but sniffed on another path by a Monitor or Router attacker will be demoted as long as the replayed path differs from the sniffed path by one link whose end node is an upgraded AS.

8. MODELING ADOPTABILITY

This section uses modeling and simulation to study the adoptability and the incremental deployment benefit of Passport. We are interested in this study because ingress filtering, despite being lightweight, offers little incentive for adoption. An AS that deploys it can still have its addresses spoofed at other parts of the network.

To examine whether Passport provides greater security benefit to motivate adoption, we use the framework presented in [10] to compare the adoptability of Passport with that of ingress filtering and SAVE [26], a protocol

to establish route-based filters [31]. We compare with SAVE because to the best of our knowledge, route-based filtering is the most effective non-cryptographic method that mitigates spoofing with partial deployment [3, 31], and SAVE is the only proposal that implements accurate route-based filters.

8.1 The Model

Our adoptability model simulates the adoption decision of each AS via iterations. At each iteration, an AS S that has not deployed a spoof prevention mechanism compares its security benefit before and after adoption. If the benefit exceeds its cost threshold, S adopts the mechanism. The iterations stop when no more ASes adopt the mechanism. The critical threshold—the largest cost threshold that leads to full deployment—measures the security benefit a spoof prevention mechanism provides to an adopter. The higher the metric, the greater the benefit, as an AS is willing to pay a higher cost to adopt it.

For an AS S that considers to adopt a spoof prevention mechanism, we define the security benefit as the average probability that an attacker cannot spoof S 's addresses in the network. To calculate this probability, an AS S iterates through every other AS D , and examines whether a malicious attacker at an AS M can spoof its addresses at D . A security indicator $E(M, D) \in \{0, 1\}$ is set to 1 if M can not spoof S , and 0 otherwise. The probability that an attacker cannot spoof S at D is computed as $\sum_M E(M, D)P(M)$, in which $P(M)$ is the probability that M is malicious. The security benefit of S , denoted by F is the weighted average of $\sum_M E(M, D)P(M)$ among all D s. That is,

$$F = \frac{\sum_D \omega_D \sum_M E(M, D)P(M)}{\sum_D \omega_D} \quad (1)$$

The weight ω_D models that an AS S sends different amount of traffic to different AS D s. Intuitively, the more traffic S sends to D , the more important that S 's addresses are not spoofed at D . The weight $P(M)$ models different security levels of different ASes.

An AS S computes the security benefit F' after its adoption, and its current security benefit F without its adoption, and uses the difference $\Delta F = F' - F$ as its incentive for adoption. At each iteration, S compares ΔF with a cost metric c . If $\Delta F > c$, it adopts the spoof prevention mechanism.

In our model, we use a uniform cost metric c for all ASes to normalize the security benefit F . Larger ASes may have higher deployment costs, but they also have larger address spaces. F can be considered as benefit per address, as it does not include the size of S .

8.2 Mechanisms

We briefly introduce ingress filtering and SAVE, and describe how to compute the security benefit F for them

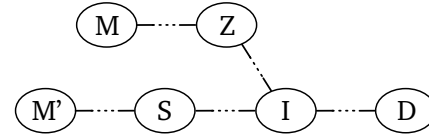


Figure 10: A sample topology.

and for Passport.

Ingress filtering: An AS S that deploys ingress filtering can filter spoofed traffic originated from hosts in its network or from its customer ASes as well as incoming traffic that spoofs its own addresses [16]. Before an AS S deploys ingress filtering, any malicious node can spoof its address space at an AS D . After S adopts ingress filtering, an attacker can still spoof S at D , unless the attacker's traffic to D is forwarded via S , or $D = S$. For instance, in Figure 10, if M' is the malicious node, after S deploys ingress filtering, M' cannot spoof S at D .

SAVE: SAVE [26] is a proposal that automatically installs route-based filters. A router maintains an incoming table that maps a source address prefix to an incoming interface at the router. A source AS that deploys SAVE periodically sends a source address update for every destination prefix in its routing table. A router uses the incoming interface of an update message to update its incoming table. When a router receives a packet, it discards the packet if its incoming interface does not match the one associated with its source address in the incoming table.

Before an AS S deploys SAVE, any malicious node can spoof S at an AS D . After an AS S deploys SAVE, a malicious node can not spoof S at D , if there is at least one upgraded AS on the path from the AS S to D , and the incoming interface of S at the upgraded AS is different from that of the malicious node. In Figure 10, if M is an attacker and Z has deployed SAVE, then M cannot spoof S at D , because S 's incoming interface to reach Z is $I \rightarrow Z$, while M 's incoming interface at Z is $M \rightarrow Z$.

Passport: If an AS S does not deploy Passport, any malicious node can spoof S 's addresses at an AS D . If an AS S deploys Passport, its security benefit depends on the attacker model. If a malicious node is a Host attacker, the malicious node cannot spoof S at D if there is at least one upgraded AS on the path from the malicious node to D . This is because the malicious node cannot spoof a valid Passport header.

If the malicious node is a Monitor attacker, it can sniff S 's traffic sent via itself, and collude with other compromised Host attackers to replay sniffed traffic. The replayed traffic by a compromised Host attacker will be demoted if the path from the Host attacker to D and the path from S to D differ by at least one link whose end node is an upgraded AS. For instance, in Figure 10, if M is a colluding compromised Host attacker, as long as the path from M to I (including I) has one AS that deploys Passport, M can not spoof S at D without being demoted,

regardless of the Monitor attacker's location. This is because an intermediate MAC in a Passport header covers the previous incoming AS number.

The security benefit under Router attacker threat is similar to that under Monitor attacker, except that a Router attacker can replay sniffed packets at its own location.

Both ingress filtering and SAVE have the same security benefit under a Monitor threat and a Host threat, because they do not insert secrets in packet headers. Under the Router attacker threat, a Router attacker on an AS S 's forwarding path can always spoof S 's addresses, before or after S adopts ingress filtering or SAVE.

We note that in computing the security benefit for Passport, as long as M 's spoofed traffic is demoted at D , S considers its traffic not spoofable by M at D , as M 's spoofed traffic is distinguishable from its authentic traffic.

8.3 Adoptability

We run simulations to compare the critical thresholds of various mechanisms. Our simulations use a generated topology of 1000 ASes, as the computational overhead is over $O(n^3)$, and we cannot finish one run on larger topologies in less than a day. For the purpose of cross validation, we use the same topology as the one used in [10]. The topology is generated using Inet [44] and BGP routing tables. We also generate smaller topologies using the same method, and run simulations on smaller topologies to confirm that the trends are consistent.

Our simulations vary the distributions of ω_D and $P(M)$, using similar models as in [10]. Due to space limitation, we only present results assuming a uniform distribution of ω_D and $P(M)$. Results using other distributions vary in the absolute values, but the trends are similar. We use 10 random initial adopters (1% of all ASes) for each run.

For the Host attacker, we simulate two different attacker populations. One assumes that a Host attacker can only be in an AS that does not deploy a spoof prevention mechanism, and the other assumes that it can be in any AS. We only show results for the first case, as the results are similar, and in the second case, Passport has stronger security benefit.

We simulate all three threat models: Host, Monitor, and Router. For Monitor and Router attackers, we assume that at most 3% of transit ASes can be compromised, and randomly pick 3% of them to be the Monitor or Router attackers. We average the security benefit over 100 runs. We assume that Host or Monitor attackers' colluding hosts can be any Host attacker.

Figure 11 shows the results. We omit the results for Router attackers for all mechanisms for clarity. Those results are very similar to the results for Monitor attackers, because in the Monitor attacker case, we already consider that a colluding Host may be on the path from S to D . As can be seen, ingress filtering has the lowest adoptability critical threshold. This is expected, because ingress

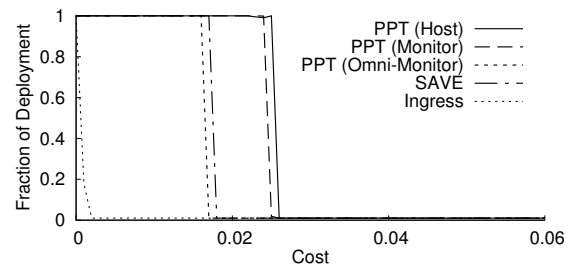


Figure 11: The steep cliff of each curve shows the critical threshold of a spoof prevention mechanism. Ingress filtering (Ingress) has the lowest adoptability. Passport (PPT) is more adoptable than SAVE under both Host and Monitor threats. Its adoptability is still close to SAVE even if we assume an omnipresent Monitor attacker.

filtering provides little immediate benefit to an adopter. Attackers can spoof an adopter's addresses at other ASes, before and after an AS's adoption.

Both Passport and SAVE are much more adoptable than ingress filtering. Passport has a higher adoptability threshold than SAVE. The adoptability threshold of Passport is only slightly affected by the presence of Monitor attackers, because 3% of Monitor attackers can only sniff a fraction of all paths. We also tested an unrealistic omnipresent Monitor model (Omni-Monitor in the figure), in which a Monitor attacker can sniff all S - D pairs. In this extreme case, the adoptability threshold of Passport is still very close to that of SAVE. This is because the algorithm of detecting replayed traffic under Passport is very similar to that of SAVE as described above.

8.4 Strong Security Benefit

Passport provides a strong security assurance that as long as an AS deploys it, other attackers cannot spoof its addresses at other ASes that also deploy it. It does not depend on transitive trusts between ASes to provide this assurance. However, the adoptability model does not capture this feature, because it computes the average probability of not being spoofed.

We define a strong security metric that measures the fraction of ASes at which no attackers can spoof an AS S 's addresses. Referring to Equation 1, when computing a strong security metric, we only include an AS D in the sum $\sum_D \omega_D \sum_M E(M, D)P(M)$, if and only if $\sum_M E(M, D)P(M) = 1$.

Figure 12 shows the strong security metric of various mechanisms under various threats. The metric is averaged over all ASes in the network. With Passport, the fraction of ASes at which no attackers can spoof a source AS's addresses scales linearly with the number of upgraded ASes. When there are Monitor or Router attackers, they may replay the Passport headers of the sources for which they provide transit service, but do not affect the security assurances of other upgraded ASes. With Router attackers, the security metric can not reach 100%

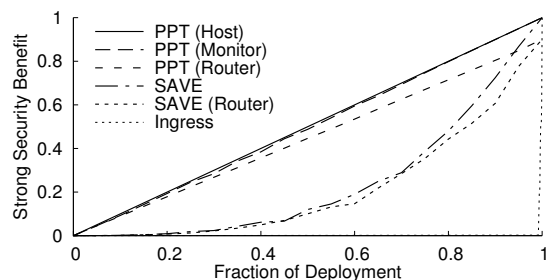


Figure 12: The fraction of ASes at which no attackers can spoof a source AS's addresses with various fractions of deployment.

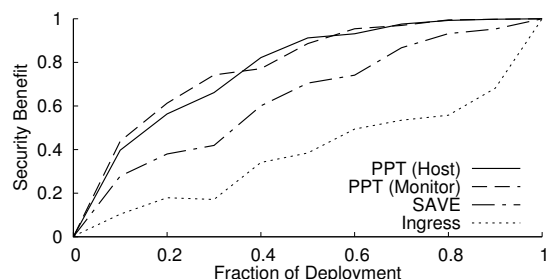


Figure 13: The security benefit F averaged over all ASes in the network with various fractions of deployment.

because on-path replayed packets can not be detected.

SAVE can not achieve this level of security assurance before it reaches a full deployment. This is because its security assurance depends on transit trust: if there is any non-upgraded AS on the path from a source to a destination, compromised hosts in that AS or connecting to that AS from a non-upgraded path can spoof the source AS's addresses.

Ingress filtering's security assurance is close to zero before it reaches full deployment, because any attacker can spoof the address space of an AS S , unless the attacker's provider deploys ingress filtering, or S provides transit service for the attacker. Our topology does not have customer-provider ingress filtering information. When computing the security benefit, we assume a leaf node's next hop AS filters spoofed traffic from the leaf node.

For comparison, Figure 13 shows the security metric F averaged over all ASes in the network using Eq 1. Again, for clarity we omit the results under Router attackers for Passport as they are similar to the ones under Monitor attackers. As can be seen, SAVE has a much higher security metric than the results in Figure 12. This suggests that the main security benefit of SAVE comes from incrementally reducing the probability of spoofing, while a large portion of Passport's security benefit is to prevent spoofing with certainty. The curves are not smooth because of the randomness in initial deployment.

Interestingly, Figure 13 shows that ingress filtering also has security benefit with partial deployment. However, this benefit does not motivate adoption, because it is al-

most the same for an AS before or after its adoption.

9. DISCUSSION

9.1 Demotion vs Discard

Passport demotes rather than discards packets with invalid MACs at intermediate ASes. Another design choice is to discard those packets. Discard has the advantage that packets will not further consume the network's resources, and legacy ASes do not need to make configuration changes. The drawback is to introduce unnecessary packet loss during routing convergence.

We choose demote over discard because a small loss rate may adversely affect TCP's performance, and presently BGP converges slowly. This design choice may not be optimal if these conditions change.

9.2 Per-Prefix Key versus Per-AS key

Our design uses per-AS key rather than per-prefix key for scalability. The memory overhead of per-prefix key is the number of prefixes one AS announces multiplying the total number of prefixes announced in BGP, a number much larger than that of per-AS key. The time it takes to recompute all shared keys after an AS re-keys is also longer. Per-AS key complicates issues such as multi-origin prefixes (as we'll describe shortly). If in the future, a router's CPU or fast memory is not a limited resource, the design can be changed to use per-prefix key.

9.3 Additional Deployment Issues

MTU Discovery: When Passport is deployed in the *bump in the wire* mode, a legacy host may not subtract the Passport header in its MTU discovery process. One solution is for a border router to intercept the ICMP "Fragmentation Needed" message, and subtract the Passport header, a practice used in the deployment of IPv4 to IPv6 [30]

Packet Fragmentation: Packets fragmented in the middle of the network will not have a valid Passport header. Passport demotes all fragments, including at the destination AS. We are not concerned much with this issue, because fragmentation by the network is discouraged, and has been disabled by IPv6.

Prefix aggregation: If an AS's prefix is aggregated by its provider AS, then its AS path attributes, including its Diffie-Hellman value will be lost. In this case, an AS should rely on its provider to stamp and verify its traffic. A customer AS that desires to stamp and verify Passport headers on its own should request its providers not to aggregate its prefix. As an AS only needs one prefix to distribute the key exchange information, even if that prefix is not aggregated, it will not significantly increase the BGP table size, as there are much more prefixes than ASes in the Internet (>244K versus <27K).

Multi-origin prefixes: BGP advertisements may have multi-origin AS conflicts (MOAS), a practice not recom-

mended by IETF [19]. MOAS interferes with the key lookup process, as a router needs to use the correct key from the origin AS to verify a Passport header. MOAS can be a signal of prefix hijacking. In this case, Passport relies on the routing system to resolve MOAS conflicts.

MOAS can be caused by sibling ASes announcing each other's prefixes [28]. In this case, they should share Diffie-Hellman values so that a verifier can use the key shared with either AS to verify their addresses.

MOAS can also be caused by multi-homed ASes connecting to its providers without BGP [51]. In this case, a simple solution is for the site to run BGP, in order to be compatible with Passport and IETF's recommendation.

Our observation from a BGP table obtained from the Oregon RouteViews server on Aug 1st, 2007 shows that 1385 out of 244095 (0.57%) prefixes have more than one origin. Therefore, we expect that MOAS prefixes are uncommon, and are unlikely to be a deployment hurdle.

Anycast addresses have legitimate multi-origins. Passport treats packets with anycast source addresses as legacy traffic. A source should not send traffic with anycast source address. An anycast address can be used as a destination address, because when computing a MAC for a destination AS, a source AS uses the key shared with the anycast address' destination AS, as described in § 3.2.

Path Discrepancy: Mao et al. [28] observe that paths inferred by traceroute may be different from BGP paths when routing is stable. They postulate several causes. Other than prefix aggregation and MOAS, most of them are due to traceroute not accurately reflecting forwarding paths or AS boundaries. One rare cause is an iBGP misconfiguration of a transit AS. Passport can become a diagnosis tool to such routing anomalies. If a router discovers that all traffic it forwards cannot be verified, it is a strong indication of misconfiguration.

Inter-domain multicast. Passport only provides source authentication for unicast traffic, because the origin of a duplicated multicast packet does not match its source address. Routers should use separate authentication mechanisms such as [34] to authenticate multicast traffic.

Enable Passport on High Speed Routers. The performance of our prototype implementation is insufficient for high speed routers. However, with optimized hardware implementation, throughput of tens of gigabits per second may be achieved. The bottleneck of Passport header processing is the AES-based MAC computation. There are already commercially available hardware AES implementation that can encrypt at the speed of 40Gbps [20]. In addition, the design of high speed routers may be revised to take into account the Passport header.

10. RELATED WORK

Router Filters: This approach maintains filter tables at routers to discard spoofed packets, and does not modify packet headers. Ingress filtering [16], route-based filter-

ing [26, 31], reverse path filtering [3], IDPF [14], and hop-count filtering [21] all fall into this category. As shown in § 8, ingress filtering provides little incentive for adoption. Route-based filtering involves non-trivial control overhead to update filter tables, and the control messages themselves need to be signed [26]. Reverse path filtering does not completely prevent address spoofing with asymmetric routing. IDPF binds a source address prefix to all incoming interfaces from which the prefix advertisement is received. It allows spoofing if an attacker's packets come from one of those interfaces, and does not work with asymmetric routing if an AS does not announce its prefixes to all providers. Hop-count filtering uses TTL heuristics to identify packets with spoofed source addresses, but attackers can still forge packets with spoofed addresses if they are no further from a destination than the sources they spoof.

Path Marking: This approach uses router-inserted path identifiers to approximate source locators [2, 45, 48, 49]. Unlike Passport, downstream routers cannot validate the authenticity of path identifiers stamped by upstream routers.

Traceback: Various traceback proposals aim to discover the sources of attack packets from router marks [38, 41, 47], or router state [40], or control messages [6]. Traceback may discover packet sources after packets are received, but does not detect spoofed packets at forwarding time like Passport.

Challenge-Response: IP or overlay routers can use connection cookies [9, 24] to validate the source address of a connection before forwarding a connection setup packet to end systems. However, this mechanism does not prevent spoofed packets from congesting a link before they reach the cookie generator.

Other cryptographic approaches. These approaches insert secrets into packets to authenticate the source address. Spoofing Prevent Method (SPM) [4] uses a 32-bit secret key shared between a source and a destination AS to authenticate the source address of a packet, and is vulnerable to eavesdropper attacks. Passport differs from SPM in that it includes a key distribution protocol (while SPM does not), and uses keyed-MACs rather than plain-text keys so they cannot be transferred to other paths, and provides the defense-in-depth needed for DoS prevention by enabling ASes in the network to verify Passport headers. We have also implemented and evaluated Passport.

Perlman's work on sabotage-proof routing uses public-key digital signatures [33] to authenticate packet sources. Our design is geared to use more efficient symmetric key MACs and hence differs in many respects.

Visa [15], SIFF [46], TVA [49], the ticket system [32], Fastpass [43], and Platypus [35] use secrets in packet headers as capabilities to authorize packets to reach a destination. Passport uses secrets to authenticate the source addresses. Capability based systems can use Passport to verify the source addresses of the capability requests.

An early design of Passport was presented in [27]. This work improves [27] and makes it more practical by considering deployment issues and reducing unnecessary features for simplicity. It also provides an implementation, evaluation, and modeling study of Passport.

11. CONCLUSION

We present and evaluate Passport, a system that allows source addresses to be validated within the network. Passport uses efficient symmetric-key cryptography to place tokens on packets that allow each AS on the path to verify that a source address is valid. ASes obtain the symmetric keys via a Diffie-Hellman key exchange piggybacked in routing messages. Passport is incrementally deployable without upgrading hosts. We have implemented Passport, evaluated it, and run experiments on the Deterlab to demonstrate its usefulness in mitigating reflector attacks. We have also analyzed its security assurances and modeled its adoptability. Our performance evaluation shows that a software PC-based router can stamp or verify Passport headers at up to 2Gbps assuming an average packet size of 400 bytes [1]. Our adoptability modeling shows that Passport provides strong security benefits to drive its adoption, and is more adoptable than ingress filtering [16] and route-based filtering [26, 31]. Together, these results suggest that cryptography-based source address authentication is feasible and advantageous.

Acknowledgment

We thank Theodore Krovetz for his help on UMAC security, and Adrian Perrig and Debabrata Dash for providing the test topology for our adoptability study, and Nanogers and Steve Bellovin for answering questions on router clock synchronization, and anonymous reviewers and other members in our group for valuable feedback.

References

- [1] CAIDA Report. http://www.caida.org/analysis/AIX/plen_hist/, 2000.
- [2] K. Argyraki and D. Cheriton. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *USENIX*, 2005.
- [3] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704, 2004.
- [4] A. B. Barr and H. Levy. Spoofing Prevention Method. In *IEEE Infocom*, 2005.
- [5] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456, 2006.
- [6] S. Bellovin. ICMP Traceback Messages, 2000. draft-bellovin-itrace-00.txt.
- [7] R. Beverly and S. Bauer. The Spoofer Project: Inferring the Extent of Source Address Filtering on the Internet. In *USENIX SRUTI*, 2005.
- [8] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO '99*, 1999.
- [9] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies Along Trust-Boundaries (CAT): Accurate and Deployable Flood Protection. In *USENIX SRUTI*, 2006.
- [10] H. Chan, D. Dash, A. Perrig, and H. Zhang. Modeling Adoptability of Secure BGP Protocols. In *ACM SIGCOMM*, 2006.
- [11] Deterlab. <http://www.deterlab.net/>.
- [12] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 1976.
- [13] D. Magoni and J.J. Pansiot. Analysis of the Autonomous System Network Topology. *ACM SIGCOMM CCR*, 31(3), July 2001.
- [14] Z. Duan, X. Yuan, and J. Chandrashekar. Constructing Inter-Domain Packet Filters to Control IP Spoofing Based on BGP Updates. In *IEEE Infocom*, 2006.
- [15] D. Estrin, J. C. Mogul, G. Tsudik., and K. Anand. Visa Protocols for Controlling Inter-Organization Datagram Flow. *IEEE JSAC*, 1989.
- [16] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing. RFC 2827, May 2000.
- [17] A. Greenhalgh, M. Handley, and F. Huici. Using routing and tunneling to combat DoS attacks. In *USENIX SRUTI*, 2005.
- [18] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *USENIX NSDI*, 2005.
- [19] J. Hawkinson and T. Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930, 1996.
- [20] Helion Technology. AES Cores. <http://www.heliontech.com/aes.htm>, 2008.
- [21] C. Jin, H. Wang, and K. G. Shin. Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic. In *ACM CCS*, 2003.
- [22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOCS*, 18(3), Aug. 2000.
- [23] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, 2006.
- [24] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Taming IP Packet Flooding Attacks. In *Proc. HotNets-II*, 2003.
- [25] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. *Lecture Notes in Computer Science*, 1751:446–265, 2000.
- [26] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. SAVE: Source Address Validity Enforcement. In *IEEE INFOCOM*, 2002.
- [27] X. Liu, X. Yang, D. Wetherall, and T. Anderson. Efficient and Secure Source Authentication with Packet Passports. In *USENIX SRUTI*, 2006.
- [28] Z. M. Mao, J. Rexford, J. Wang, and R. Katz. Towards an Accurate AS-Level Traceroute Tool. In *ACM SIGCOMM*, 2003.
- [29] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, 1998.
- [30] E. Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). RFC 2765, 2000.
- [31] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *ACM SIGCOMM*, 2001.
- [32] B. Patel and J. Crowcroft. Ticket Based Service Access for the Mobile User. In *ACM MobiCom*, 1997.
- [33] R. Perlman. Network Layer Protocols with Byzantine Robustness. MIT Ph.D. Thesis, 1988.
- [34] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient and Secure Source Authentication for Multicast. In *NDSS*, 2001.
- [35] B. Raghavan and A. C. Snoeren. A System for Authenticated Policy-Compliant Routing. In *ACM SIGCOMM*, 2004.
- [36] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006.
- [37] RouteViews Project. <http://www.routeviews.org/>.
- [38] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, 2000.
- [39] F. Scalzo. Anatomy of recent DNS reflector attacks from the victim and reflector points of view. Nanog37 Presentation, June 2006.
- [40] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountios, S. Kent, and W. Strayer. Hash-Based IP Traceback. In *ACM SIGCOMM*, 2001.
- [41] D. Song and A. Perrig. Advance and Authenticated Marking Schemes for IP Traceback. In *Proc. IEEE Infocom*, 2001.
- [42] R. Thomas. ISP Security BOF, NANOG 28. <http://www.nanog.org/mtg-0306/pdf/thomas.pdf>, 2003.
- [43] D. Wendlandt, D. G. Andersen, and A. Perrig. FastPass: Providing First-Packet Delivery. Technical report, CMU-CyLab, 2006.
- [44] J. Winick and S. Jamin. Inet-3.0: Internet Topology Generator. Technical Report CSE-TR-456-02, University of Michigan, 2002.
- [45] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend Against DDoS Attacks. In *IEEE Symposium on Security and Privacy*, 2003.
- [46] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *IEEE Symposium on Security and Privacy*, 2004.
- [47] A. Yaar, A. Perrig, and D. Song. FIT: Fast Internet Traceback. In *Proc. of IEEE Infocom*, 2005.
- [48] A. Yaar, A. Perrig, and D. Song. StackPi: New Packet Marking and Filtering Mechanisms for DDoS and IP Spoofing Defense. *IEEE JSAC*, 2006.
- [49] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *ACM SIGCOMM*, 2005.
- [50] K. Zetter. Cisco Security Hole a Whopper. <http://www.wired.com/politics/security/news/2005/07/68328>, 2005.
- [51] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. An Analysis of BGP Multiple Origin AS (MOAS) Conflicts. In *ACM IMW*, 2001.

Context-based Routing: Techniques, Applications and Experience

Saumitra Das¹, Yunnan Wu², Ranveer Chandra², Y. Charlie Hu¹

¹Purdue University, ²Microsoft Research, Redmond

Abstract

Routing protocols in multi-hop networks typically find low cost paths by modeling the cost of a path as the *sum of the costs* on the constituting links. However, the insufficiency of this approach becomes more apparent as new lower layer technologies are incorporated. For instance, to maximize the benefit of multiple radios, ideally we should use routes that contain low interference among the constituting links. Similarly, to maximize the benefit of network coding, we should ideally use routes that create more coding opportunities. Both of these are difficult to accomplish within the conventional routing framework because therein the links are examined in isolation of each other, whereas the nature of the problem involves interdependencies among the links.

This paper aims at revealing a unifying framework for routing in the presence of inherent link interdependencies, which we call “context-based routing”. Through the case studies of two concrete application scenarios in wireless networks, network coding-aware routing and self-interference aware routing in multi-radio systems, we highlight the common key pillars for context-based routing and their interplay: a context-based path metric and a route selection method. We implement context-based routing protocols in Windows XP and Linux and evaluate them in detail. Experiments conducted on 2 testbeds demonstrate significant throughput gains.

1 Introduction

Routing in wireless mesh networks is a well studied problem. A common practice is to model the cost of a path as the sum of the costs on the constituting links, where the link cost reflects the link quality, e.g., the per-hop round-trip time (RTT) [1], the expected transmission count (ETX) [2], and the expected transmission time (ETT) [4]). Routing then aims at finding the path offering the lowest total cost. However, the inadequacy of this widely used approach becomes evident as new lower layer technologies appear.

For example, a promising technique for improving the capacity of wireless mesh networks is to use multiple radios. With multiple radios, more concurrent communications can be packed in spectrum, space and time. To maximize the benefit of multiple radios, ideally we should

use routes that contain low interference among the constituting links. Another example is link layer network coding, a recent advance [6] that exploits the broadcast nature of the wireless medium. Network coding, on its own, can improve the link layer efficiency. The gain of this technique, however, critically depends on the traffic pattern in the network, and hence the routing decisions. To maximize the benefit of network coding, ideally we should use routes that create more mixing opportunities. In these scenarios, the conventional routing framework does not perform well because therein the links are examined in isolation of each other. To fully leverage the lower layer advances, we need advanced routing techniques that can properly model the inherent interdependencies of the links in order to identify good routes.

Related Work: For multi-radio systems, some progress has been made in selecting interference aware routes. The WCETT [4] (Weighted Cumulative Expected Transmission Time) metric penalizes a path that uses the same channel multiple times, thus modeling link interference to some extent. However, WCETT assumes that all links on a route that use the same channel interfere, which does not incorporate the phenomenon of spatial reuse and may miss high-throughput routes that reuse channels carefully at links sufficiently apart (see Figure 4). Moreover, [4] uses Dijkstra’s shortest path algorithm. Although Dijkstra’s algorithm is optimal for the conventional path metric, i.e. a sum of link costs, it is not optimal for WCETT. Another proposal for interference aware routing is the MIC (metric of interference and channel-switching) metric [13, 14] and the authors also showed how to find the optimal route under this metric in polynomial time. However, to ensure finding the optimal path in polynomial time, the metric is forced with some constraints (decomposability, fixed memory) that may not match well with the nature of the underlying link interdependencies. This can cause issues in modeling the costs of routes (see Figure 5), and as a result, the path found can be inefficient.

Network coding-aware routing has been studied in [8, 11] from theoretical perspectives. These papers presented theoretical, flow-based formulations for computing the throughput with network coding and network coding-aware routing, assuming centralized control in routing and scheduling. To characterize the gain of net-

work coding, these theoretical studies showed that it is necessary to examine two consecutive hops jointly, because of the link interdependency.

Overview: These previous studies have offered insight in dealing with link interdependencies. This paper aims at revealing a unifying framework for routing under inherent link interdependencies, which we call “context-based routing”. We proceed by studying two concrete application scenarios: network coding-aware routing, and self-interference aware routing. Through these case studies, we highlight the common key pillars for context-based routing and their interplay, and provide a more systematic treatment of these pillars. In particular, we show how to overcome the shortcomings of the aforementioned approaches for self-interference aware routing in multi-radio systems and also apply the techniques more generally to other scenarios.

The first pillar is the concept of *context-based link metrics*, which can model the interactions among different links in a route. Context refers to examining the cost of each link based on what links are used prior to the current link. Having such context allows us to evaluate the “goodness” of routes by considering the impact of the links on each other. More specifically, the conditional link costs allow us to conveniently characterize effects such as: (i) the self-interference caused by other links of the same flow, and (ii) the reduction of channel resource consumption due to the use of network coding.

The second pillar is an optimized route selection module, which can search for good paths under a context-based link metric. The interdependencies among the links in a context-based link metric make it challenging to search for a good path. To make things worse, sometimes the context-based link metric is by nature globally coupled, meaning that the cost of a link can depend on as far as the first link. We propose a general context-based path pruning method (CPP). To model the context of a partial path while avoiding the exponential growth in the number of partial paths, CPP maintains a set of paths that reach each node, corresponding to different *local contexts*. Each local context can be viewed as a concise summary of the dominating history of all the partial paths that end with that local context¹, based on the observation that older history usually has less impact on the future. The best path under each local context is maintained and considered further for possible expansion into a *source-destination* path.

The use of local contexts in path pruning is synergistic with the use of contexts in cost modeling. Together, these two pillars form a context-based routing protocol (CRP), which outperforms existing approaches in scenarios where modeling link-interdependencies is critical.

¹e.g. sequence of channels or links used in the previous l hops

The essence and key technical contribution of CRP lies in (i) properly modeling the link interdependencies (via the context-based metric) and (ii) handling the ensuing algorithmic challenges in route optimization (via CPP).

We have implemented CRP on Windows XP and Linux. While our current implementation assumes link-state routing, other generalizations are also possible. Our evaluations show that CRP provides TCP throughput gains up to 130% and on average around 50% over state-of-the-art approaches for multi-radio networks and gains of up to 70% over state-of-the-art approaches for wireless network coding.

The rest of the paper is organized as follows: Section 2 defines context-based routing metrics and their application to two scenarios. Section 3 describes a general technique of context-based path pruning for finding routes under context-based routing metrics. Section 4 evaluates the performance of the overall context-based routing solution. Finally, Section 5 concludes the paper.

2 Context-Based Routing Metrics

In this section, we discuss what context-based routing metrics are and how they help in modeling link interdependencies. The common defining feature of context-based metrics is that the cost of a link is context-dependent (dependent on what links are already part of the route). To make things concrete, we consider two specific systems: multi-radio multi-channel networks and single-radio networks equipped with network coding and show how context-based metrics can better match the nature of the problems.

2.1 ERC: Context-based metric for exploiting network coding

In this section, we define a context-based metric that can help make better routing decisions in wireless networks equipped with network coding.

In *network coding*, a node is allowed to generate output data by mixing (i.e., computing certain functions of) its received data. The broadcast property of the wireless medium renders network coding particularly useful. Consider nodes v_1, v_2, v_3 on a line, as illustrated in Figure 1. Suppose v_1 wants to send packet x_1 to v_3 via v_2 and v_3 wants to send packet x_2 to v_1 via v_2 . A conventional solution would require 4 transmissions in the air (Figure 1(a)); using network coding, this can be done using 3 transmissions (Figure 1(b)) [12]. It is not hard to generalize Figure 1 to a chain of nodes. For packet exchanges between two wireless nodes along a line, the consumed channel resource could potentially be halved.

Katti et al. [6] recently presented a framework for network coding in wireless networks, in which each node

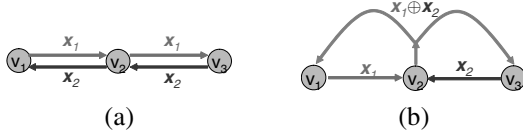


Figure 1: Network coding example.

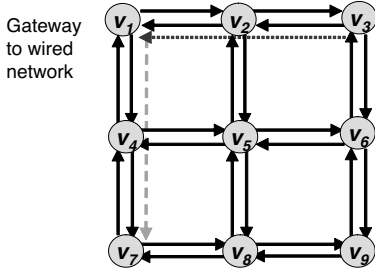


Figure 2: An example mesh network. There are 9 mesh access points and v_1 is a gateway to the wired network.

snoops on the medium and buffers packets it heard. A node also informs its neighbors which packets it has overheard. This allows nodes to know roughly what packets are available at each neighbor (i.e., “who has what?”). Knowing “who has what” in the neighborhood, a node examines its pending outgoing packets and decides how to form output mixture packets, with the objective of most efficiently utilizing the medium. These prior studies result in a link layer enhancement scheme in the networking stack. The network coding engine sits above the MAC layer (e.g., 802.11) and below the network layer. Given the routing decisions, the network coding engine tries to identify mixing opportunities. The gain of this technique, however, critically depends on the traffic pattern in the network. This motivates the following question: Can intelligent routing decisions maximize the benefits offered by network coding?

A natural thought is to modify the link metrics to account for the effect of network coding in reducing transmissions. This, however, is not straightforward. Consider the example setting illustrated in Figure 2. There are two long-term flows in the network, $v_3 \rightarrow v_2 \rightarrow v_1$ and $v_1 \rightarrow v_4 \rightarrow v_7$. We want to find a good routing path from v_1 to v_9 . Due to the existence of the network coding engine, the route $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_6 \rightarrow v_9$ is a good solution because the packets belonging to this new flow can be mixed with the packets belonging to the opposite flow $v_3 \rightarrow v_2 \rightarrow v_1$, resulting in improved resource efficiency. To encourage using such a route, can link $v_2 \rightarrow v_3$ announce a lower cost? There are some issues in doing so, because a packet from v_5 that traverses $v_2 \rightarrow v_3$ can not share a ride with a packet from v_3 that traverses $v_2 \rightarrow v_1$, although a packet from v_1 that traverses $v_2 \rightarrow v_3$ can.

Thus, in the presence of the network coding engine, assessing the channel resource incurred by a packet

transmission requires some context information about where the packet arrives from. For example, we can say that given the current traffic condition, the cost for sending a packet from v_2 to v_3 , that previously arrives from v_1 , is smaller. The key observation here is the need to define link cost based on some context information². Specifically, for this application, we model the cost of a path $\mathcal{P} = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ as the sum of the conditional costs on the links:

$$\text{cost}(\mathcal{P}) \triangleq \text{cost}(v_0 \rightarrow v_1) + \text{cost}(v_1 \rightarrow v_2 | v_0 \rightarrow v_1) + \dots + \text{cost}(v_{k-1} \rightarrow v_k | v_{k-2} \rightarrow v_{k-1}). \quad (1)$$

Here $\text{cost}(b \rightarrow c | a \rightarrow b)$ denotes the cost of sending a packet from b to c , conditioned on that the packet arrived from a . The central issue is to properly define the link costs and compute them. Let us begin by reviewing a conventional (unconditional) link metric. A popular link quality metric in the literature is the expected transmission count (ETX) [2]. This metric estimates the number of transmissions, including retransmissions, needed to send a unicast packet across a link. The ETX metric is a characterization of the amount of resource consumed by a packet transmission.

We now describe how to define a conditional link metric to model the resource saving due to network coding. With network coding, several packets may share a ride in the air. Naturally, the passengers can share the airfare. In effect, each participating source packet is getting a discount. Such a discount, however, cannot be accurately modeled by an unconditional metric such as ETX, because the applicability of the discount depends on the previous hop of the packet. We propose a conditional link metric called the *expected resource consumption* (ERC), which models the cost saving due to network coding. Consider a packet sent in the air. If it is a mixture of k source packets, then each ingredient source packet is charged $\frac{1}{k}$ the resource consumed by the packet transmission. The resource consumed could be measured in terms of, e.g., air time, or consumed energy.

Computation of Expected Resource Consumption (ERC)

We now explain how to concretely compute the ERC. Each node maintains a `WireInfoTable`. Each row of the table contains the measured statistics about a wire. A wire is a virtual link that connects an incoming and outgoing link from a node (connected the past and future hop), e.g. $e_{i,j} \rightarrow e_{j,k}$, which crosses the current node v_j . A cost of a wire $e_{i,j} \rightarrow e_{j,k}$ represents the conditional cost $\text{cost}(j \rightarrow k | i \rightarrow j)$. The packets forwarded by the current node can be classified into categories associated with the wires. A packet that is received along $e_{i,j}$ and sent along $e_{j,k}$ falls into the category “ $e_{i,j} \rightarrow e_{j,k}$ ”.

²For network coding, limiting context to the previous hop suffices.

For each wire category, we collect the total number of packets sent and the total resource consumed in a sliding time window. The total resource consumption is obtained by adding the resource consumption for each sent packet. A simple charging model is used in our current implementation. For example, if a source packet across wire $e_{i,j} \rightarrow e_{j,k}$ is sent in a mixture of 3 packets, we set the resource consumption of this source packet as $1/3$ of the ETX of link $e_{j,k}$. (We could also use ETT [4] in lieu of ETX.)

To implement the sliding window computation efficiently, we quantize the time axis into discrete slots of equal length. We use a sliding window of N slots. For each wire, we maintain a circular buffer of N bins; at any time, one of the N bins is active. At the start of each slot, we shift the active bin pointer once and clear the statistics in the new active bin. Each time a packet is transmitted in the air, we update the statistics in the current active bin accordingly. We use $N = 10$ slots, each of length 0.5s.

To evaluate the conditional link metric for a certain wire $e_{i,j} \rightarrow e_{j,k}$, we first obtain the ERC for each slot, say n , as: $\text{erc}_n := \frac{\text{Resource consumed by pkts sent in slot } n}{\# \text{ of packets sent in slot } n}$. Then we compute the ERC for the wire as the weighted average of the ERCs for the slots:

$$\text{ERC} := \sum_{n=0}^{N-1} \alpha_n \text{erc}_n; \quad \alpha_n = \alpha^{N-1-n} \left(\frac{1 - \alpha}{1 - \alpha^N} \right). \quad (2)$$

Here the parameter α is the forgetting factor for old observations. Older observations receive lower weights. In the simulations, we use $\alpha = 0.8$.

The above measurement method generates an estimate of the *current ERC*, which is the ERC seen by a flow whose packets are *currently being mixed*. To bootstrap new flows to favor routes that expose mixing opportunities, in addition to the current ERC, we also collect another statistic called the *marginal ERC*. The marginal ERC reflects the potential new ERC, i.e., with discount, for a wire if a new flow decides to use a route that contains that wire. A flow decides on its initial route or switches to a new route based on the marginal ERC. Afterwards, the current ERC for each wire in the chosen route is updated and is used instead. If the existing flows already use up most of the mixing opportunities, then the marginal ERC will not have a high discount. Both the current ERC and the marginal ERC are reported. To compute the marginal ERC, a simple rule is applied in our current implementation. Specifically, for a wire $e_{ij} \rightarrow e_{jk}$ in a given time slot, we examine the number of unmixed packets y in the reverse wire $e_{kj} \rightarrow e_{ji}$. If $y \geq 25$, then we set the marginal ERC as 0.75 of the ETX (25% discount); otherwise, we set the marginal ERC as the ETX (no discount).

Dealing with oscillation ERC takes the traffic load into account. Could this cause oscillation in the routing decisions? The advertised discounts are conditional by definition; hence they typically only apply to a specific set of flows (those that can be mixed with the existing flows at a node). For example, node v_2 in Figure 2 might advertise a smaller conditional cost, $\text{cost}(v_2 \rightarrow v_3 | v_1 \rightarrow v_2) < \text{cost}(v_2 \rightarrow v_3)$, to reflect that traffic going v_1 to v_2 and then v_3 can be mixed with the existing flows. Such conditional discounts can only be taken advantage by flows that uses the segment $v_1 \rightarrow v_2 \rightarrow v_3$. There is incentive for this specific set of flows to route in a certain cooperative manner that are mutually beneficial. Presumably, if the flows try such a mutually beneficial arrangement for some time, they will confirm the discounts and tend to stay in the arrangement. To prevent potential route oscillations, we require each flow to stay for at least T_{hold} duration after each route change, where T_{hold} is a random variable. The randomization of the mandatory route holding time T_{hold} is used to avoid flows from changing routes at the same time. In addition, after the mandatory route holding duration, the node switches to a new route only if the new route offers a noticeably smaller total cost.

In summary, using context conveniently represents paths that benefit from network coding. We now show another example of a context-based metric for networks with multiple-radios.

2.2 SIM: Context-based metric for exploiting multiple radios

In this section, we define a context-based metric that can help make better routing decisions in wireless networks equipped with multiple radios.

Consider a multi-hop wireless network equipped with multiple radios. Similar to [4], we assume each radio is tuned to a fixed channel for an extended duration; a route specifies the interfaces to be traversed. We define a context-based SIM (self-interference aware) metric, as a weighted sum of two terms:

$$\text{SIM}(\mathcal{P}) \triangleq (1 - \beta) \sum_k \text{ETT}(e_k) + \beta \max_k \text{ESI}(e_k | \mathcal{P}_{k-1}). \quad (3)$$

where e_k is the k -th edge along the path \mathcal{P} .

The first term is the sum of the expected transmission time along the route. The ETT [4] metric aims at estimating the average transmission time for sending a unit amount of data. It is defined as: $\text{ETT} \triangleq \frac{\text{PktSize} \cdot \text{ETX}}{\text{Link Bit-Rate}}$. ETX [2] is computed as $\text{ETX} \triangleq 1/(1-p)$, where p is the probability that a single packet transmission over link e is not successful. The link bit-rate can be measured by the

technique of packet pairs and this method can produce sufficiently accurate estimates of link bandwidths [3].

ESI of the Bottleneck Link The second term calculates the estimated service interval (ESI) at the bottleneck link, which reflects how fast the route can send packets in the absence of contending traffic. Note that the max operation is used here instead of the sum operation. This can be explained by a pipeline analogy. In a pipeline system consisting of several processing stages, the long term throughput is determined by that of the bottleneck stage, rather than the average throughput of the stages. Sending packets along a wireless route is similar. The long term throughput of a flow is determined by that of the bottleneck link.

The ESI of a link is defined as:

$$\text{ESI}(e_k|\mathcal{P}_{k-1}) \triangleq \text{ETT}(e_k) + \sum_{j < k} p_{jk} \text{ETT}(e_j). \quad (4)$$

Here \mathcal{P}_{k-1} is the partial path with $k - 1$ links, and p_{jk} is a binary number that reflects whether e_j and e_k interfere. Characterizing the interference relations among the links is itself a research challenge. For instance, one method is to make use of actual interference measurement; see, e.g., Padhye et al. [9] and the references therein. Consider two links, $A \rightarrow B$ and $C \rightarrow D$, using the same channel. As suggested in [9], for 802.11 networks, the primary forms of interference are four cases: (i) C can sense A 's transmission via physical or virtual carrier sensing, and hence refrains from accessing the medium, (ii) A can sense C 's transmission, (iii) transmissions of A and C collide in D , (iv) transmissions of A and C collide in B . Based on this, we adopt a simplified approach in the experiments of this paper. We treat the two links as interfering if A has a link to C or D with sufficiently good quality, or C has a link to A or B with sufficiently good quality.

The ESI expression (4) leaves out the interference caused by other contending traffic. This is a simplification in modeling. The ESI expression (4) considers the self-interference from the previous hops of the route, by adding up the expected transmission times of the previous links. The intuition is that the packets at the link needs to share the channel with the interfering links on the route. One might ask why we do not add the ETTs from the subsequent links on the path, even though both previous and subsequent links can create interference. The following theorem provides an answer.

Theorem 1 (Interpretation of Bottleneck ESI)

Assuming ideal scheduling, sufficiently long flow, absence of contending traffic, and an ideal binary interference model dictated by a conflict graph, the end-to-end throughput of $1/\max_k \text{ESI}(e_k|\mathcal{P}_{k-1})$ is achievable.

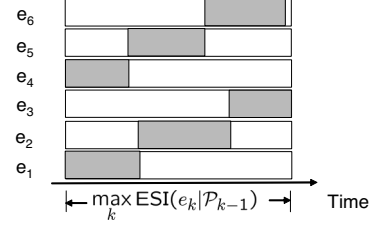


Figure 3: An interference-free scheduling of links. Assume e_k interferes with $e_{k-1}, e_{k-2}, e_{k+1}, e_{k+2}$. A shaded region for a link e_k indicates that e_k is using the medium.

Proof: Under the assumptions in the claim, finding the optimal end-to-end throughput essentially amounts to finding an optimal interference-free scheduling of the uses of the constituting links. If we can schedule each link to transfer B bits in T seconds, then the throughput B/T can be achieved. It is well known that this problem can be viewed as a continuous version of the graph coloring problem on the conflict graph.

In greedy coloring algorithms, nodes in a graph are visited one by one. Each node tries to reuse some existing colors if possible. If not, the node selects a new color. With this procedure, it is easy to see that the graph can be colored in $\Delta(U) + 1$ colors, where $\Delta(U)$ is the maximum degree of a vertex. Notice that the greedy coloring algorithm always look at the already colored nodes, but not future nodes. Hence in fact the upper-bound can be tightened to one plus the maximum number of already-colored neighbors for the nodes. We now apply a greedy-coloring like algorithm for scheduling the links on a route. This is illustrated in Figure 3 for a path with 6 links. We visit the links on the route sequentially, from the first hop to the last hop. For each link e_k , find one or more intervals with a total length of $\text{ETT}(e_k)$ that do not cause interference with any of the previous $k - 1$ links. Similar to greedy coloring, when assigning the intervals to a link, we only need to examine the previous links, but not future links. With this greedy scheduling process, we can finish the assignment in a total duration of $\max_k \text{ESI}(e_k|\mathcal{P}_{k-1})$. If we repeat this scheduling pattern for a sufficiently long time, then we can deliver one packet end to end every $\max_k \text{ESI}(e_k|\mathcal{P}_{k-1})$ (sec). Hence the throughput is achievable. ■

The above theorem shows that the bottleneck ESI corresponds to a theoretically achievable throughput. Conversely, if a link e_k interferes with a set \mathcal{F} of previous links, then typically links in $\mathcal{F} \cup \{e_k\}$ would be expected to mutually interfere (hence forming a clique in the conflict graph). If that indeed is the case, then we cannot deliver more than one packet end to end every $\max_k \text{ESI}(e_k|\mathcal{P}_{k-1})$ (sec). This argument shows that the maximum throughput is roughly around $1/\max_k \text{ESI}(e_k|\mathcal{P}_{k-1})$.

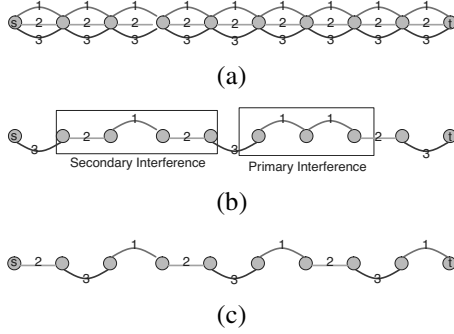


Figure 4: (a) Chain topology. Each node has 3 radios on orthogonal channels 1, 2, 3. (b) Route selected by using the WCETT metric. (c) Route selected by using the SIM metric.

Related Work: Comparison with Other Metrics
The bottleneck ESI models self-interference. The previously introduced metrics, WCETT and MIC, also consider self-interference. We now compare the proposed SIM metric with WCETT and MIC.

The WCETT metric proposed by Draves et al. [4] is defined as:

$$\text{WCETT}(\mathcal{P}) \triangleq (1 - \beta) \cdot \sum_{e_k \in \mathcal{P}} \text{ETT}(e_k) + \beta \cdot \max_{\text{Channel } j} X_j, \quad (5)$$

$$X_j \triangleq \sum_{e_k \text{ is on channel } j} \text{ETT}(e_k).$$

Here e_k denotes the k -th hop on the path \mathcal{P} and β is a weighting factor between 0 and 1. The WCETT metric is a weighted sum of two terms. The first term is the sum of the ETT along the path. The second term aims at capturing the effect of self-interference. A path that uses a certain channel multiple times will be penalized.

Although WCETT considers self-interference, it has some drawbacks. Consider the following example. Figure 4(a) shows a chain topology, where each node has three radios tuned to orthogonal channels 1, 2, 3. We assume two links within 2 hops interfere with each other if they are assigned the same channel. We further assume all links have similar quality. An ideal route in this setting is then a route that alternates among the three channels, such as the one illustrated by Figure 4(c), because it can completely avoid self-interference. Figure 4(b) shows a possible solution returned by WCETT. This route suffers from primary interference as well as secondary interference. This can be explained by the way WCETT models self-interference. From (5), it is seen that WCETT views a path as a set of links. In this example, it tries to maximize channel diversity by balancing the number of channels used on the path. Thus the path in Figure 4(b) and the path in Figure 4(c) appear equally good under WCETT.

In essence, WCETT takes a pessimistic interference model that all links on the same channel in the route

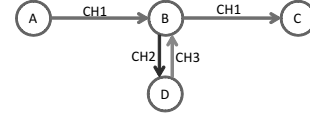


Figure 5: Under the MIC metric, it is possible that the route $A \rightarrow B \rightarrow D \rightarrow B \rightarrow C$ has a lower cost than the route $A \rightarrow B \rightarrow C$. Edges are labeled with the channels.

interfere with each other. Indeed, if we use this interference model, then SIM reduces to WCETT. The proposed SIM metric models self-interference can differentiate different ordering of the links in the path and find the optimal path for this example. The benefit is that SIM potentially allows better channel reuse.

Another self-interference aware metric is the metric of interference and channel-switching (MIC), proposed by Yang et al. [13, 14]. The MIC metric is defined as:

$$\text{MIC}(\mathcal{P}) \triangleq \alpha \sum_{e_k \in \mathcal{P}} \text{IRU}(e_k) + \sum_{\text{Node } i \in \mathcal{P}} \text{CSC}_{i_i}, \quad (6)$$

$$\text{IRU}(e_k) \triangleq \text{ETT}(e_k) \times N_k, \quad (7)$$

$$\text{CSC}_i \triangleq \begin{cases} w_1 & \text{if previous hop is on a different channel,} \\ w_2 & \text{otherwise. } 0 \leq w_1 < w_2. \end{cases} \quad (8)$$

Here $\alpha > 0$ is a weighting factor to balance the two terms. In (7), N_k is the number of neighbors in the network that interferes with the transmission of link e_k . The IRU (interference-aware resource usage) term aims at reflecting the inter-flow interference. A link that interferes with more neighbors will be penalized. The CSC (channel-switching cost) aims at reflecting the self-interference, since it penalizes consecutive uses of the same channel.

The total CSC term (8) in the MIC metric models the self-interference by considering the immediate previous hop. As an extension, the authors also considered the extension of the MIC metric to model self-interference for more than one hops [14]. However, a potential limitation with the MIC metric and its multi-hop extension is that it has a fixed limited memory span. Consider the example in Figure 5. Suppose the links between nodes A and B have very low costs. Under the MIC metric, it is possible that the route $A \rightarrow B \rightarrow D \rightarrow B \rightarrow C$ has a lower cost than the route $A \rightarrow B \rightarrow C$. Since the path is selected by optimizing the metric, it is unclear whether the MIC metric can rule out the possibility of selecting a pathological path, which has self-interference that are not modeled in the MIC expression.

Compared with the MIC metric, the SIM metric considers the possible interference with all previous hops. This avoids the pathological case as shown in Figure 5. In addition, whereas the link CSCs are summed up in (6), SIM uses the maximum ESI. The use of the maximum

operation can better reflect the fact that the throughput is determined by the bottleneck link.

3 Context-Based Path Pruning

The previous section showed two concrete examples of how context helps in making routing metrics more powerful. After defining a good context-based path metric, the subsequent challenge is to find the optimal (or near-optimal) route under the path metric. A context-based routing protocol needs both a context-based metric and a way to find good paths under such metrics. In this section, we develop such a path finding method called CPP (Context-based Path Pruning) that can find paths under any generic context-based metric. As a starting point, we first consider a link state routing framework. In link state routing, each router measures the cost to each of its neighbors, constructs a packet including these measurements, sends it to all other routers, and computes the shortest paths locally. Hence for link state routing, what is needed is a centralized algorithm for computing the shortest paths. The CPP method will be explained as a centralized algorithm. But we note that it can also be applied in some distributed settings, such as distance vector protocols and on-demand route discovery; these extensions are omitted in the interest of space.

Let us begin by reviewing the (simpler) problem of finding the optimal path under a path metric where each link has a nonnegative cost and the cost of a path is the sum of the costs of the constituting links. This problem is well understood. For example, the classical shortest path algorithm by Dijkstra can be applied to find the optimal path with complexity $O(|V|^2)$, where $|V|$ denotes the number of nodes in the network. Dijkstra's algorithm maintains upper-bound labels $f(v)$ on the lengths of minimum-cost $s-v$ paths³ for all $v \in V$. The label of each node is either *temporary* or *permanent*, through the execution of the algorithm. At each iteration, a temporary label with the least total cost is made permanent and the remaining temporary labels are updated. Specifically, if v^* has the minimum total cost among the temporary nodes, then we update the cost of every other temporary node w as:

$$f(w) := \min \{f(w), f(v^*) + c(v^*w)\}. \quad (9)$$

Dijkstra's algorithm operates on an optimality principle: The shortest path from s to t via v is the shortest path from s to v concatenated with the shortest path from v to t . Thus, each node only needs to remember the cost of the best $s-v$ path. Such optimality principle no longer holds for metrics such as SIM, ERC or WCETT; for such

metrics, an $s-v$ path \mathcal{P}_1 may have a larger cost than an $s-v$ path \mathcal{P}_2 but may eventually lead to a lower cost toward the final destination node t .

We propose a context-based path pruning (CPP) technique as a heuristic method for optimizing a context-based metric. To model the potential impact of past hops on future hops, we maintain a *set* of paths that reach each node v , instead of a single $s-v$ path with minimum cost. A natural question is: How many paths should we store at each node as we search for a good $s-t$ path? Storing all paths would apparently result in an exponential complexity. To keep the complexity manageable, we maintain only a small subset of $s-v$ paths at a node v ; the size of the subset is constrained by the affordable computation complexity. Ideally we want this small set to cover all "promising" $s-v$ paths, which have the potential of leading to an optimal $s-t$ path. How do we select this set then? Consider dividing an $s-t$ path into three parts as:

$$s \xrightarrow{\mathcal{P}_1} u \xrightarrow{\mathcal{P}_2} v \xrightarrow{\mathcal{P}_3} t. \quad (10)$$

A good route in a wireless mesh network typically moves in the direction toward the destination. If the second part \mathcal{P}_2 is sufficiently long, typically the first segment $s \xrightarrow{\mathcal{P}_1} u$ would be well separated from the third segment $v \xrightarrow{\mathcal{P}_3} t$; as a result, there would not be a strong interdependency between the first and the third segment. In other words, the link interdependencies of a wireless route are typically localized in nature. This observation motivates us to organize the memory at each node according to several *local contexts*. As a concrete example, we can define the local context of an $s-v$ path as the sequence of links in the last l hops, where l is a predetermined parameter. Under this definition, if \mathcal{P}_2 in (10) is l -link long, then the local context of the $s-v$ path $s \xrightarrow{\mathcal{P}_1} u \xrightarrow{\mathcal{P}_2} v$ is the sub-path \mathcal{P}_2 . The $s-v$ paths with the same local contexts are grouped together. During the execution of the proposed algorithm, node v always keeps only one $s-v$ path under each local context, i.e., the minimum cost one found so far in the algorithm execution. The algorithm works by examining the paths stored at the nodes and considering them for potential expansion into an $s-t$ path.

In the above we have given a concrete example definition of the local context, where the local context of an $s-v$ path is defined as the sequence of links in the last l hops, where l is a predetermined parameter. For the scenario of network coding-aware routing, we use this definition with $l = 2$. As another concrete example, we can define the local context of an $s-v$ path as the sequence of channels taken by the links in the last l hops. This definition is useful for the scenario of self-interference aware routing. In general, the definition of local contexts is problem specific.

More formally, Algorithm 1 shows a Dijkstra-style in-

³An $s-v$ path refers to a path that begins at s and ends at v .

Algorithm 1 A Context-based Path Pruning Method

INPUT: A function that can evaluate the cost of a path.
 $T := \{s\}$; /* The set of temporary paths. */
 $P := \emptyset$; /* The set of permanent paths. */
while $T \neq \emptyset$ **do**
 choose the path \mathcal{P}^* from T with the minimum cost;
 $T := T - \mathcal{P}^*$; $P := P + \mathcal{P}^*$;
 for each valid extension of \mathcal{P}^* , $\mathcal{P} = \mathcal{P}^* + e$, **do**
 $c := \text{LocalContext}(\mathcal{P})$; /* The definition of local contexts
 is problem specific. Two examples are: the sequence of links
 in the last l hops, and the sequence of channels taken by the
 links in the last l hops. */
 if $T \cup P$ contains a path Q with local context c **then**
 replace Q by \mathcal{P} if it has a lower cost than P ;
 else
 $T := T + \mathcal{P}$;
 end if
 end for
end while
OUTPUT: For each node v , find the best local context $c^*(v)$ result-
ing in minimum cost. For each context c of each node v , store the
best link reaching it with minimum cost. To recover a route from v
to s , back-track from $c^*(v)$ along the best links toward s .

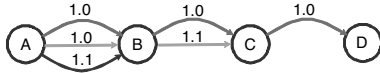


Figure 6: An example graph. There are three orthogonal channels, CH1, CH2, CH3, from top to bottom. The edges are labeled with their ETT.

stantiation of the CPP method. We maintain a set T of temporary paths and a set P of permanent paths⁴. In each step, we choose the temporary path with minimum cost. Such a path, say \mathcal{P}^* , is made permanent. Then we consider the possible ways of extending \mathcal{P}^* toward an s - t path. For each extension $\mathcal{P} = \mathcal{P}^* + e$, we determine its local context and search for a path with the same local context in T and P . If a path with the same local context already exists, then such existing path is compared with \mathcal{P} and the winner is retained. If a path with the same local context does not exist, then \mathcal{P} is added to T .

There is an alternative way to understand the CPP method. For each physical node v , introduce one vertex v_c for each local context c applicable to v and interconnect the vertices according to original connectivity. Denote such a *context-expanded graph* by G_c . We can interpret Algorithm 1 as applying a revised version of the Dijkstra's algorithm to the expanded graph G_c . More specifically, since here the path metric is not decomposable, the cost update step (9) needs to be revised. Instead of using (9), node v^* first reconstructs the current best path from the source, say $s \xrightarrow{\mathcal{P}} v^*$. Then each neighbor

⁴Permanent and temporary paths are concepts used in the path-finding algorithm. These notions are extended from the original Dijkstra's algorithm. In Algorithm 1, temporary paths (reaching a specific context) are still subject to improvement (in terms of cost minimization) in the algorithm execution. In each step, the temporary path with minimum cost is made permanent. Only permanent paths will be considered for expansion into a longer path.

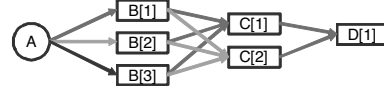


Figure 7: Expanded graph when the local contexts are defined by the previous hop's channel ID. $X[i]$ corresponds to the local context where the incoming link is on channel i .

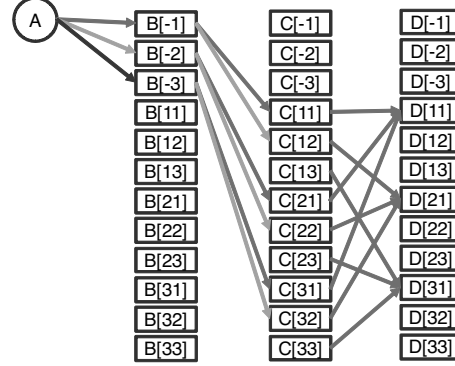


Figure 8: Expanded graph when the local contexts are defined by the previous two hops' channel IDs. Here $X[ij]$ corresponds to the context where this node is reached via a link in channel i , followed by a link in channel j . $X[-j]$ refers to the local context where this node is directed reached from the source via a link in channel j .

node w of v^* is updated using the following path-based update rule:

$$f(w) := \min \left\{ f(w), \text{cost} \left(s \xrightarrow{\mathcal{P}} v^* \rightarrow w \right) \right\}, \quad (11)$$

where $\text{cost}(\cdot)$ returns the path metric.

We now start from the network in Figure 6 and show how to construct context-expanded graphs for it. Here there are three orthogonal links from A to B , two links from B to C , and one link from C to D . The ETT metrics for the links are shown on the links. Consider $\beta = 0.5$ and the SIM metric (3). In this example, Dijkstra's algorithm using the path-based update rule (11) will return $A \xrightarrow{\text{CH1}} B \xrightarrow{\text{CH2}} C \xrightarrow{\text{CH1}} D$, with a total cost of $0.5 * (1.0 + 1.1 + 1.0) + 0.5 * \max\{1.0 + 1.0, 1.1\} = 2.55$.

Figure 7 shows the expanded graph for the case the local contexts are defined by the previous hop's channel ID. Taking the original node B as an example, there are three nodes in G_c , which are associated with three channels to reach B . To connect Algorithm 1 with running Dijkstra's algorithm using the path-based update rule (11) over the expanded graph, we can view each node as storing the optimal path reaching it from the source; the optimal path can be obtained by backtracking along the best links that reach each node. In this case, the optimal route found is $A \xrightarrow{\text{CH2}} B \xrightarrow{\text{CH1}} C \xrightarrow{\text{CH1}} D$, with a total cost of $0.5 * (1.0 + 1.0 + 1.0) + 0.5 * \max\{1.0 + 1.0, 1.0\} = 2.5$.

If the local contexts are defined by the previous two hops' channel IDs, then the resulting expanded graph would be the one shown in Figure 8(c). This will yield

the optimal route $A \xrightarrow{CH^3} B \xrightarrow{CH^2} C \xrightarrow{CH^1} D$, with a total cost of $0.5 \cdot (1.1 + 1.1 + 1.0) + 0.5 \cdot \max\{1.1, 1.1, 1.0\} = 2.1$.

Handling the ERC metric is simpler for CPP since ERC simple has one-hop of memory. Thus the local context is simply defined as the previous hop on which the packet arrived instead of specifying the channel. This shows that CPP is a generic technique which can be used for many context-based metrics by defining the local contexts in accordance with the definition of the metric being used.

Optimality If the path metric indeed has a fixed memory span (say, l hops) such as in ERC, then CPP with the local context defined by the l -hop links is guaranteed to find a route that minimizes the given path metric (because no pruning step is suboptimal). In our case, the SIM path metric has a memory span that could potentially involve the entire path history. Even if the path metric has a longer memory span than the length of the local contexts, the CPP method can still be applied as an effective heuristic method.

Related Work: Comparison with the Route Selection Method in [13, 14] Yang *et al.* [13, 14] proposed a method for finding the optimal route under the MIC metric (6) and its multi-hop generalization in polynomial time complexity. The method hinges upon the fact that the path metric is decomposable into a sum of link costs, where the cost of a link depends only a fixed number of previous hops; in other words, the path metric is additive and has a fixed memory span. In [13, 14], a virtual graph is constructed, by introducing virtual vertices and edges to represent different states; each edge has an associated cost. Then finding an optimal route under the decomposable, finite-memory metric in the original problem becomes the problem of finding an optimal path in the virtual graph under a memoryless metric, where the cost of a path is simply the sum of the costs of the constituting edges. Therefore, for the additive path metric with a fixed memory span, the optimal solution can be found in polynomial time.

In comparison, the CPP method is applicable for all path metrics, not only when the path metric is decomposable and has finite memory. A key feature of the CPP method is the explicit differentiation of two memories: the path metric's memory and the local context's memory. The path metric's memory can be chosen to best model the path cost. For instance, the path metric based on ERC has 2-hop memory; the SIM path metric has global memory. However, the local context's memory will be chosen based on complexity. These two memories need not be equal length. When the path metric is decomposable and has finite memory, CPP can also find the optimal answer, by using a local context with the

same memory length as the path metric. When the path metric is not decomposable, CPP can still be used, as a heuristic method. This comes from the fact that CPP operates by always examining the partial-paths from s and applying the path metric to evaluate their costs, which is apparent from Algorithm 1, as well as the path-based update rule in (11). Observe also that the edges in Figures 7 and 8 do not have associated costs, in contrast to the virtual graph in [13, 14]. Although the local context used in CPP only has limited information about the path history, the path metric has the information of the complete path and can properly take into account any local or global link interdependency. This mix of "local" memory and "global" path evaluation metric is a distinct feature of the CPP method, rendering it generally applicable and efficient.

Complexity As we mentioned earlier, Algorithm 1 can be essentially viewed as applying Dijkstra's algorithm with the path-based update rule (11) over the expanded graph. Note though that in Algorithm 1, the involved vertices and links are constructed on the fly, without explicitly maintaining the expanded graph. Due to the connection between Algorithm 1 and Dijkstra's algorithm, we can easily conclude that the time complexity of the Algorithm 1 is $O(C^2)$, where C is the total number of local contexts at all nodes. If we define the local context of a path as the sequence of channels taken by the links in the last l hops, then C is upper-bounded by $(K + K^2 + \dots + K^l) \cdot |V(G)|$, where K is the number of channels in the system and $V(G)$ is the set of nodes in the original network. For practical purposes, we specifically propose to use $l = 2$; see Figure 8 for an example definition of the local contexts. This would lead to a specific complexity of $O((K + K^2) \cdot |V|)^2$.

4 Performance

This section documents our experience with the Context Routing Protocol using simulations and a real deployment. CRP was implemented on both WindowsXP as well as Linux.

4.1 Evaluation Methodology

Simulation Setup We implemented CRP in QualNet 3.9.5, a widely used simulator for wireless networks. CRP was implemented with support for multiple interfaces, ETX and packet pair probing for ETT calculations, periodic dissemination of link metrics, the WCETT, SIM and ERC metrics and the CPP route selection method. The simulations use the 802.11a MAC and a two-ray fading signal propagation model. All radios operate at a nominal physical layer rate of 54 Mbps and support autorate. We use both UDP and TCP flows in the evaluation. The simulations use 2-hop context length for SIM

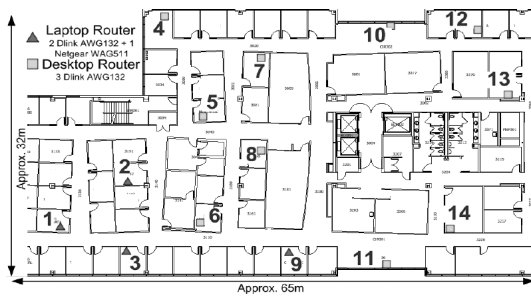


Figure 9: The topology of the wireless testbed A.

and 1-hop context length for ERC. We also implemented a complete version of COPE network coding [6] in Qual-Net as a shim layer based on the protocol description.

Testbed Setup We implemented CRP on Windows XP and Linux and tested it on two separate wireless testbeds. CRP for Windows was implemented by extending the MR-LQSR protocol [4] implementation as a loadable Windows driver that sits at layer 2.5. CRP appears like a single virtual network adapter to applications by hiding the multiple physical interfaces bound to it. This allows unmodified applications to run over CRP. Routing operates using 48 bit virtual Ethernet addresses of the MCL adapter in each node. This choice for our CRP implementation allows a direct and fair comparison with the WCETT metric since it is based on the same underlying codebase. The CRP code uses 2-hop channel ids as context. CRP on Linux was implemented by extending the SrcRR routing protocol from the RoofNet project. We also used the publicly available COPE implementation from the authors of the protocol to implement network coding. This implementation was available only for Linux. CRP disseminates link metric information periodically similar to MR-LQSR. For multi-radio routing, CRP introduces no additional control overhead in link-state packets. The only cost is the additional memory required for computation of routes which is not significant even for large networks with lot of radios. For network-coding aware routing, CRP piggybacks additional information on link-state updates (i.e. the conditional metrics). This results in slightly larger update packets when network coding opportunities are available.

The CRP protocol was deployed on two testbeds: (1) A 14 node wireless testbed running Windows XP (network A), and (2) A 20 node wireless testbed running Mandrake Linux (network B). Network A is located on a floor of an office building in Redmond and is illustrated in Figure 9. 10 of the nodes are small form factor HP desktops each with 3 DLink AWG132 802.11a/b/g USB cards while 4 nodes are Toshiba tablet PCs with one Netgear WAG511 802.11a/b/g PCMCIA card and 2 DLink AWG132 802.11a/b/g cards. The driver configurations are modified to allow multiple cards from the same vendor to co-exist in a machine. The radios on each

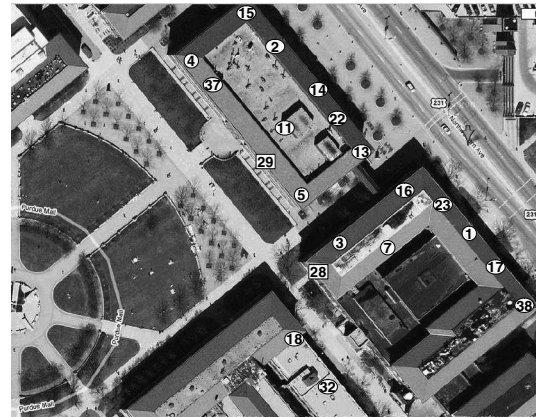


Figure 10: The topology of the wireless testbed B.

node have their own SSID and are statically assigned the 802.11a frequencies 5180, 5240 and 5300 GHz. Thus, the 42 radios form three 14-node networks each with its own frequency and SSID glued together through the CRP virtual adapter. Each radio works in ad-hoc mode and performs autorate selection. The OS on each machine implements TCP-SACK. Finally, the nodes are static and use statically assigned private IPv4 addresses. The results are expected to not be significantly affected by external interference since no other 802.11a network was in the area. Network B is located across 3 academic building at Purdue University and is shown in Figure 10. The machines are small form factor HP desktops each with an Atheros 802.11a/b/g radio and run Linux.

4.2 Simulation Results

We first demonstrate the performance of CRP in a controlled and configurable simulator setting.

4.2.1 CRP in a Frequency Reuse Scenario

The first scenario demonstrates how CRP can exploit frequency reuse opportunities in a correct way. We consider a chain of 10 nodes separated by 300m, each with 3 radios on 3 orthogonal channels. The transmission power used causes interference to nodes even two hops away which is typical in real networks. Figure 11 shows the routes selected by using WCETT (on top) and CRP (below).

Notice the route selected by WCETT: while it maximizes channel diversity, i.e., each channel is used exactly equally (3 times), it cannot optimize the ordering of the channel use (as explained in Section 2.2) while CRP chooses the optimal route. How does this route selection impact performance? Figure 11 also shows the UDP and TCP throughput achieved using ETT, WCETT and CRP on this chain topology. The results show that CRP route selection has a significant impact on performance improving UDP throughput by 166% over WCETT and

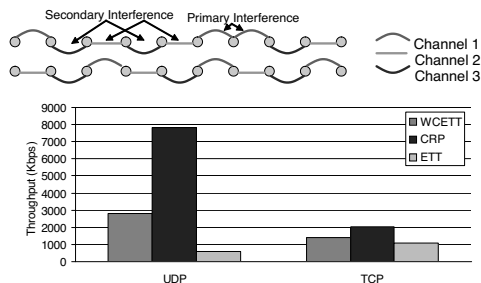


Figure 11: Performance under frequency reuse.

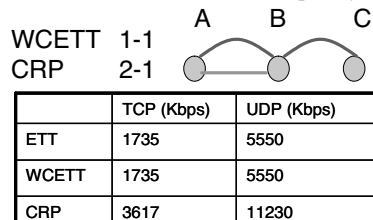


Figure 12: Performance under heterogeneous nodes.

over 1000% compared to ETT. For TCP, CRP improves the throughput by around 44% and 91% with respect to WCETT and ETT, respectively. Thus, CRP is able to select a path with high raw capacity. However, the gains are reduced due to TCP's known inability to reach the available bandwidth effectively in multi-hop wireless networks (which is also amplified by the long chain). An interesting aspect to note here is that the benefit comes from using the SIM metric and either CPP or Dijkstra's algorithm would choose the correct route. However, the next section shows when SIM by itself is not enough and demonstrates the usefulness of context-based path selection.

4.2.2 CRP in a Heterogeneous Node Scenario

In this scenario, we consider a simple 3 node topology as shown in Figure 12. Node A and B have 2 radios tuned to channel 1 and 2 while node C has one radio tuned to channel 1. Even in such a simple topology, WCETT and ETT are unable to choose the good route which can simply be decided by observation. This is because of the 'forgetfulness' of Dijkstra. Once the route to B is decided simply based on the lowest metric to get to B, it cannot be changed. Thus, both ETT and WCETT in MR-LQSR choose the path with both hops on channel 1 while CRP chooses the path with channels 1 and 2 since CRP does not finalize the interface used to reach B right away. Figure 12 shows that the CRP route improves the TCP throughput by 108% and UDP throughput by 102% over both WCETT and ETT. Note that the performance improvement is similar for TCP and UDP since in a two-hop scenario, TCP is better able to utilize the available bandwidth.

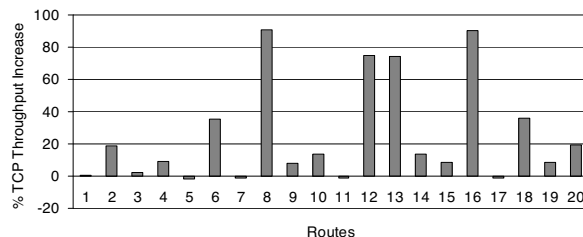


Figure 13: Performance in a large network.

4.2.3 CRP in a Large Network

We now evaluate CRP in a large network of 50 static nodes randomly placed in a 2000 m x 2000 m area. We selected 20 random non-overlapping source destination pairs in the network and initiated a TCP connection between each pair for 60 seconds (so as to observe steady state behavior). Each node had 3 radios on orthogonal channels.

The results in Figure 13 shows the percentage increase in TCP throughput of CRP over WCETT. We find that 7 paths have 20% or more gain in TCP throughput with the gains being as high as 90% in some cases. Typically whenever the performance of CRP and WCETT are equal, it is for two reasons: (1) The path is 3 hops or shorter in length. Since WCETT assigns channels in equal proportions, it always chooses a route similar to CRP (each link has a different channel), (2) Although rare, sometimes when the path is longer than 3 hops, by random chance the WCETT ordering of channels turns out to be optimal.

4.2.4 CRP in Network Coding Scenarios

We now evaluate CRP for a single-radio network with network coding in QualNet. Consider the network shown in Figure 2 with an existing flow $v_3 \rightsquigarrow v_1$. After 3 seconds, v_1 initiates a flow to v_9 . There are many possible routes that this flow can take, but only one ($v_1 - v_2 - v_3 - v_6 - v_9$) is optimal in terms of the resource consumption. CRP causes the flow $v_1 \rightsquigarrow v_9$ to choose the mutually beneficial route $v_1 - v_2 - v_3 - v_6 - v_9$, resulting in maximized mixing. Intuitively, the existing flow $v_3 \rightsquigarrow v_1$ creates a discount in terms of the conditional ERC metric in the opposite direction, which attracts v_1 to choose route $v_1 - v_2 - v_3 - v_6 - v_9$. Once the flows start in both directions, they stay together and mix because both see discounts. As shown in Table 1, CRP increases the number of mixed packets in this scenario by 12 \times in comparison to LQSR+COPE.

We continue with the 9-node grid network scenario and evaluate the performance with three flows: (1) $v_9 \rightsquigarrow v_1$, (2) $v_1 \rightsquigarrow v_9$, and (3) $v_3 \rightsquigarrow v_1$. Each flow begins randomly between 50–60 seconds into the simulation. We evaluate the performance of LQSR, LQSR+COPE and

Scenario	Mixed (CRP+COPE)	Mixed (LQSR+COPE)
S1	20,366	1,593
S2	39,576	24,197

Table 1: Gain from CRP+COPE compared to LQSR+COPE.

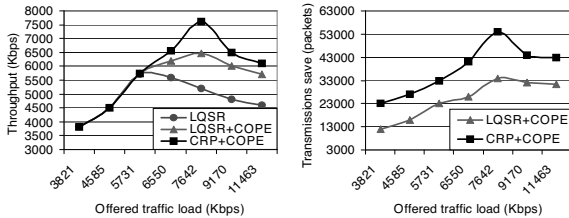


Figure 14: Throughput comparison and transmissions saved with CRP+COPE, LQSR+COPE and LQSR.

CRP+COPE for this scenario for different input loads. The results are depicted in Figure 14. CRP provides throughput gains compared to LQSR (up to 47%) and LQSR+COPE (up to 15%). Without CRP, flows mix essentially by chance. In the example, the flow 1-2-3-6-9 is mixed with 9-6-3-2-1 with CRP, due to the mutually beneficial discounts enjoyed by both flows.

Figure 14 also gives the amount of *resource* saved by using CRP. CRP consistently provides reduction of packet transmissions of over 10,000 packets across a wide variety of traffic demands. Another observation from Figure 14 is that the saved transmissions reduce as the network load increases. This is counter-intuitive since more packets should indicate more mixing opportunities. However, this occurs because of the *capture* effect in the 802.11 MAC layer which is amplified at high loads. Due to this, packets from only one node (the capturing node) fill queues for large durations of time without allowing other traffic. This reduces mixing opportunities at high load. This problem can potentially be addressed through a better MAC layer design that avoids capture.

Thus, the simulation results for multi-radio networks show that gains from CRP are observed in two cases: (1) When frequency reuse is possible, and (2) When there are heterogeneous nodes in the network, i.e. not all nodes have the same number of interfaces. While these scenarios are common and important to address (typical wireless networks are bound to have some frequency reuse opportunities and networks may have all different types of nodes with different number of radios and characteristics), our testbed evaluation in the next section provides insight into other scenarios where CRP is helpful. In systems with network coding, CRP performance exemplifies that network coding itself cannot provide the best achievable performance and a protocol such as CRP can help network coding achieve better performance.

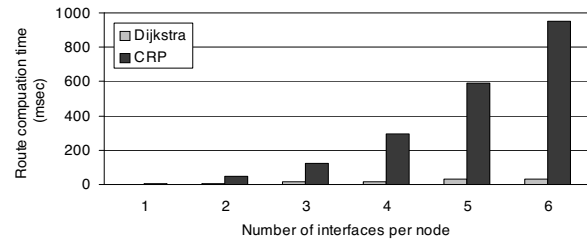


Figure 15: Microbenchmark: route computation time.

4.3 Testbed Evaluation

We now document our experience with the performance of CRP in a real wireless network deployment. We first perform a series of microbenchmarks to see its performance in known settings and then evaluate its performance as a running system.

4.3.1 Computational Complexity

CRP improves Dijkstra route selection at the expense of more computation and complexity. While the complexity is not a hurdle (the protocol is already implemented), it is important to find the computational complexity to ascertain the protocols applicability to real mesh networks which may have embedded devices with slow processors.

We configured our CRP implementation's graph cache with a graph topology of 100 nodes (taken from a typical neighborhood layout) and varied the number of interfaces per node. We then evaluated the time taken by CRP versus Dijkstra in computing the shortest paths to all nodes in the network. We find that our implementation, while significantly more computationally expensive than Dijkstra, can find routes in the extreme case of 100 node networks with 6 radios each in 900 ms. The current MCL implementation in fact calls Dijkstra's algorithm only once per second, caching routes in between. However in our current testbed with 14 nodes each with 42 radios, the computation is performed in around 10-20 ms. Thus, even if we use embedded mesh routers whose CPU speeds are 5-6 times slower than our testbed nodes, we expect to typically find routes under 100 ms. Additionally, one could proactively find such routes in the background to hide this delay as well.

4.3.2 Frequency Reuse Scenario

We now evaluate the real performance gains from good choice of routes in our testbed. Note that unlike the simulator which does not simulate adj-channel interference⁵, this gives us a better idea of the potential performance benefits.

We selected one of the 4 hops routes in testbed A between nodes 12 and 9 and performed multiple 1-minute

⁵QualNet physical layer code is binary only so we could not change it to simulate adj-channel interference

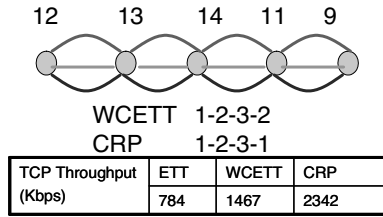


Figure 16: Microbenchmark: frequency reuse.

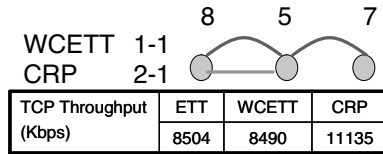


Figure 17: Microbenchmark: heterogeneous nodes.

TCP transfers taking the median performance. The route and channels chosen by WCETT and CRP are shown in Figure 16. WCETT reuses channel 2 on the last link 11→9 essentially because the channel 2 radio on node 11 had a lower ETT. Thus, WCETT took a local view not realizing this link would cause secondary interference with link 13→14. However, CRP chose the interface on channel 1 on the last link which although locally had a higher ETT, did not cause self-interference. Thus this route got a higher SIM score and was selected by CRP providing a performance gain of 60% and close to 200% with respect to WCETT and ETT respectively. This example clearly shows how using context benefit translates into real world gain.

4.3.3 Heterogeneous Node Scenario

We next consider the simple topology of 3 to see how much gain a CRP provides over a WCETT route in the real world. As shown in Figure 17, we configured node 8 and 5 with two interfaces on channel 1 and 2 while node 7 had one interface on channel 1.

WCETT chose channel 1 on both links due to the use of Dijkstra and ETT also chose this route because the ETT of channel 2 on node 8 was around 1ms higher than channel 1. However, CRP is able to identify the correct route and chooses a route with channel 2 and channel 1. This allows CRP to have a TCP throughput 31% higher than both WCETT and ETT. Note that the gain is lower than that observed in simulation because some adj-channel interference can limit gains. From measurements, we know such interference exists despite our attempts to place the cards at some distance from each other on each machine.

4.3.4 Large Node Scenario

We now evaluate the gain from CRP in a running network topology. Each testbed node in this scenario has 3 radios.

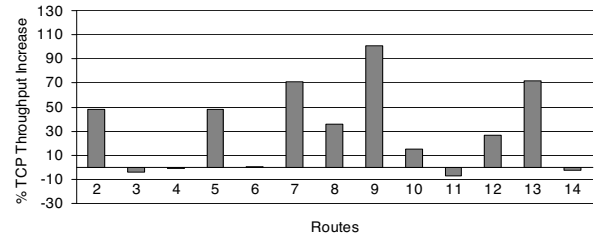


Figure 18: Testbed macrobenchmark: Basic scenario. TCP throughput gain on paths from node 1 to all other nodes.

We performed multiple 1-minute TCP transfers between node 1 and all the other testbed nodes with WCETT and CRP. The % throughput gain of CRP over WCETT observed from node 1 to the other nodes (numbered on the x-axis) is shown in Figures 18.

The results show that CRP can improve WCETT throughput in 7 out of 13 paths by up to 100% and more than 30% on most paths. Some of the results are expected while some are non-intuitive. Among the expected results are: (1) The paths to nodes 3, 6, 4 and 11⁶ have no gain since they are less than 4 hops in length and WCETT can assign costs correctly if the number of hops are less than the number of radios per node. (2) Nodes 5 and 7 are reached via 4-hop routes and the SIM metric can now choose a better path to provide gain.

Among the unexpected results are: (1) Paths to nodes 8 and 9 have gain despite their being 3 hops away. We found this was caused by ETT variation among the interfaces of node 6 which rendered 2 interfaces almost useless due to high ETTs and effectively caused a bottleneck link which required careful route selection. CRP identified and used the good interface on node 6 and thus performed better. (2) Routes to node 10 have low gain despite reuse opportunities since the link performance is constrained by the weak link 8→10. However, going to nodes 12 and 13 through 10 gives gain since the throughput is now constrained more by ordering of channels selected than the weak link. (3) Finally, although 14 is 4 hops away, there is no gain because WCETT chooses the good route selected by CRP purely by chance.

We also evaluated the gain in a heterogeneous network scenario where all nodes do not have the same number of interfaces. In these experiments, gains of up to 152% were observed.

4.3.5 Network Coding Scenario

We now perform experiments with our Linux CRP implementation on top of COPE in testbed B. We take multiple examples of the Alice-and-Bob topologies from our testbed and demonstrate how much gain is provided by

⁶Node 4 can be reached with 2 hops due to it being in an open area and the waveguide effect of hallways.

Scenarios	UDP Gain	TCP Gain
22→3, 3→22	1.66×	1.29×
18→5, 5→18	1.53×	1.24×
11→7, 7→11	1.78×	1.31×
37→13, 2→5	1.71×	1.24×

Table 2: Median throughput gain from CRP+COPE compared to LQSR+COPE for both UDP and TCP. For each scenario we initiated the flows 15 times.

simply using SrcRR+COPE versus using CRP+COPE. This testbed has only one radio on each node. We use the ERC metric in CRP. Table 2 shows the gains achieved using CRP for different topologies tested in the testbed. The left hand column lists the two flows initiated in the network in each case and the node numbers refer to Figure 10.

The table shows significant gains from CRP+COPE over SrcRR+COPE in the range of $1.7\times$ for UDP and $1.27\times$ for TCP. In the first scenario, SrcRR chooses the routes 22-13-3 and 3-5-22 which does not provide gain from network coding while causing interference among the two flows. CRP entices flow 3-22 onto the same route due to the ERC discounts and allows network coding to occur at node 13 for the duration of the transfer. In the second scenario under SrcRR, flow 18-5 uses the route 18-28-5 while flow 5-18 uses the low quality direct route 5-18. CRP advertises discounts on the 28-18 link for traffic from 5 and this causes the two flows to mix at node 28 resulting in gain. In the third scenario, SrcRR choose the routes 11-5-3-7 and 7-16-13-11/7-16-5-11 for the two flows resulting in no coding of packets while CRP brings the two flows onto a mutually beneficial route traversing nodes 5 and 3 in both directions. Finally in the fourth scenario, CRP allows nodes 37 and 2 to choose 11 as a next hop once ERC discounts become visible. Node 5 and 13 can overhear the packets from 37 and 2 respectively giving rise to network coding gain. Without CRP the two flows tend to take the routes 37-11-5 and 2-14-22 resulting in no coding gain. Thus, we can see the benefits that CRP provides to the COPE system allows the network coding engine more opportunities to code packets. Note that if a lot of flows are present in the network (such as the scenarios evaluated in [6]), the benefits from CRP would be less significant since a large number of packets present in the network can provide enough coding opportunities without intelligent route selection. However, CRP can provide COPE with coding opportunities even in more sparse or lightly loaded networks.

Note that we currently do not aim at solving the load balanced routing problem in handling multiple flows. We believe a better practical strategy is for link traffic to affect the link metric and then use a method, such as CRP, to find a route. Thus CRP should potentially be useful

for load-balanced routing proposals.

5 Conclusion

In this paper, we investigated *context-based routing*, a general route selection framework that models link interdependencies and selects good routes, through the case studies of network coding-aware routing and self-interference aware routing in multi-radio systems. The effectiveness of our approach is demonstrated through both simulations and a deployed implementation. In the future, we plan to investigate other potential applications for CRP by studying further scenarios where link interdependencies exists such as multi-radio networks with network coding or lightpath selection in WDM optical networks.

Acknowledgements

This work was supported in part by NSF grant CNS-0626703. We thank the reviewers and our shepherd Sue Moon for their comments.

References

- [1] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou. A multi-radio unification protocol for IEEE 802.11 wireless networks. In *Proc. of BroadNets*, 2004.
- [2] D. S. J. D. Couto, D. Aguayo, J. C. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proc. of ACM MobiCom*, 2003.
- [3] R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. In *Proc. of ACM SIGCOMM*, 2004.
- [4] R. Draves, J. Padhye, and B. Zill. Routing in multi-radio, multi-hop wireless mesh networks. In *Proc. of MobiCom*, Sept. 2004.
- [5] D. B. Johnson and D. A. Maltz. *Dynamic Source Routing in Ad Hoc Wireless Networks*. Kluwer Academic, 1996.
- [6] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft. Xors in the air: Practical wireless network coding. In *Proc. of ACM SIGCOMM*, August 2006.
- [7] P. Kyasanur and N. Vaidya. Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks. In *MC2R 10(1):31–43, Jan*, 2006.
- [8] B. Ni, N. Santhapuri, Z. Zhong, and S. Nelakuditi, “Routing with opportunistically coded exchanges in wireless mesh networks,” In *Proc. WiMesh’06*, Sept. 2006.
- [9] J. Padhye, S. Agarwal, V. Padmanabhan, L. Qiu, A. Rao, and B. Zill. Estimation of Link Interference in Static Multi-hop Wireless Networks. In *Proceedings of IMC*, 2005.
- [10] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proc. of ACM SIGCOMM*, August 1994.
- [11] S. Sengupta, S. Rayanchu, and S. Banerjee, “An analysis of wireless network coding for unicast sessions: The case for coding-aware routing,” in *Proc. IEEE INFOCOM*, May 2007.
- [12] Y. Wu, P. Chou, and S.-Y. Kung. Information exchange in wireless networks with network coding and physical-layer broadcast. In *Proc. of CISS*, 2005.
- [13] Y. Yang, J. Wang, and R. Kravets. Designing routing metrics for mesh networks. In *Proc. of WiMesh*, 2005.
- [14] Y. Yang, J. Wang, and R. Kravets. Interference-aware load balancing for multihop wireless networks. In *Technical Report UIUCDCS-R-2005-2526, Dept. of Comp. Science, UIUC*, 2005.

NetComplex: A Complexity Metric for Networked System Designs

Byung-Gon Chun
ICSI

Sylvia Ratnasamy
Intel Research Berkeley

Eddie Kohler
UCLA

Abstract

The systems and networking community treasures “simple” system designs, but our evaluation of system simplicity often relies more on intuition and qualitative discussion than rigorous quantitative metrics. In this paper, we develop a prototype metric that seeks to quantify the notion of algorithmic complexity in networked system design. We evaluate several networked system designs through the lens of our proposed complexity metric and demonstrate that our metric quantitatively assesses solutions in a manner compatible with informally articulated design intuition and anecdotal evidence such as real-world adoption.

1 Introduction

The design of a networked system frequently includes a strong algorithmic design component. For example, solutions to a variety of problems – routing, distributed storage, multicast, name resolution, resource discovery, overlays, data processing in sensor networks – require distributed techniques and procedures by which a collection of nodes accomplish a network-wide task.

Design simplicity is a much-valued property in such systems. For example, the literature on networked systems often refers to the importance of simplicity (all emphasis added):

The advantage of Chord is that it is *substantially less complicated*. . . (Chord [35])

This paper describes a design for multicast that is *simple to understand*. . . (Simple Multicast [32])

This paper proposed a *simple* and effective approach. . . (SOSR [17])

Likewise, engineering maxims stress simplicity:

All things being equal, the simplest solution tends to be the right one. (Occam’s razor)

KISS: Keep It Simple, Stupid! (Apollo program)

However, as the literature reveals, our evaluation of the simplicity (or lack thereof) of design options is often through qualitative discussion or, at best, proof-of-concept implementation. What rigorous metrics we do employ tend to be borrowed from the theory of algorithms. For example, two of the most common metrics used to calibrate system designs are the amount of state maintained at nodes and the number of messages ex-

changed across nodes. These metrics, however, were intended to capture the overhead or efficiency of an algorithm and are at times incongruent with our notion of simplicity. For example, flooding performs poorly on the number of messages exchanged across nodes, but most of us would consider flooding a simple, albeit inefficient, solution. Similarly, a piece of state obtained as the result of a distributed consensus protocol feels intuitively more complex than state that holds the IP address of a neighbor in a wireless network.

We conjecture that this mismatch in design aesthetic contributes to the frequent disconnect between the more theoretical and applied research on networked system problems. A good example of this is the work on routing. Routing solutions with small forwarding tables are widely viewed as desirable and the search for improved algorithms has been explored in multiple communities; for instance, a fair fraction of the proceedings at STOC, PODC, and SPAA are devoted to routing problems. The basic distance-vector and link-state protocols incur high routing state ($O(n)$ entries) but are simple and widely employed. By contrast, a rich body of theoretical work has led to a suite of *compact* routing algorithms (e.g., [2,3,10,36]). These algorithms construct optimally small routing tables ($O(\sqrt{n})$ entries) but appear more complex and have seen little adoption.

This is not to suggest existing overhead or efficiency metrics are not relevant or useful. On the contrary, all else being equal, solutions with less state or traffic overhead are strictly more desirable. Our point is merely that design simplicity plays a role in selecting solutions for real-world systems, but existing efficiency or performance-focused metrics can be misaligned with our notion of what constitutes simple system designs.

This paper explores the question of whether we can identify complexity metrics that more directly capture the intuition behind our judgment of system designs. Because the system designs we work with are fairly well-specified, we believe there is no fundamental reason why our appreciation of a design cannot be reinforced by quantifiable measures. Such metrics would not only allow us to more rigorously discriminate between design options, but also to better align the design goals of the theory and systems communities.

We start by reporting on a survey we conducted to understand how system designers evaluate and articulate complexity in system design (Section 2). Building on

this, we define a complexity metric in Sections 3 and 4 and evaluate several networked system designs through the lens of our complexity metric in Section 5. Using this analysis, we demonstrate that our metric quantitatively differentiates across flavors of solutions and ranks systems in a manner that is congruent with our survey. We discuss the limitations of our metric in Section 6, review related work in Section 7, and conclude in Section 8.

Finally, it is important to clarify the scope of our work. We intend for our complexity metric to complement – not replace – existing efficiency or performance metrics. For example, in the case of a routing algorithm, our metric might capture the complexity of route construction but reveal little about the quality of computed paths. In addition, while we focus on system design at the algorithmic or procedural level, there are many aspects to a software system that contribute to its ultimate complexity. For example, as the CAP theorem [13] tells us, the careful selection of a system’s service model profoundly impacts complexity. The same is true for the sound design of its software implementation. Although at least as important as distributed complexity, these are not aspects we consider in this paper. Lastly, we stress that we view our metric as a prototype: one specific metric that works well with several classes of important systems. We expect that the best-suited metric will emerge in time after much broader discussion and evaluation (similar to the development of standard benchmarks in many communities such as databases and computer architecture). As such, we view our contribution primarily in getting the ball rolling by providing a candidate metric and set of results for further scrutiny.

2 Perceived Complexity

We conducted a survey to explore how system designers perceive complexity of networked system algorithms such as routing, distributed systems, and resource discovery. Nineteen students in a graduate distributed systems class at UC Berkeley participated in the survey. Participants were asked to rank which of two comparable networked system algorithms they viewed as more complex on a scale where 1 means system A is far more complex and 9 means B is far more complex. Participants were also asked to rationalize their choice in 2–3 sentences.

We discuss the algorithms we surveyed in detail in the later sections of this paper; Table 1 briefly summarizes the findings from the survey’s quantitative ranking. The one-sample *t*-test reveals that participants consider distance vector (DV) routing as more complex than link state (LS) routing but less complex than landmark or compact routing. In evaluating classical distributed systems, participants viewed solutions such as quorums, Paxos, multicast, and atomic multicast as more complex than read-one/write-all, two phase commit, gossip, and

Algorithm A	Algorithm B	More complex algorithm
DV	LS	A ($p < .060$)
DV	Landmark	B ($p < .050$)
DV	Compact	B ($p < .030$)
DV	RCP	not significant
Read one/write all	Quorum	B ($p < .007$)
Two phase commit	Paxos	B ($p < .001$)
Gossip	Multicast	B ($p < .013$)
Atomic multicast	Repeated multicast	A ($p < .001$)
Locking	Lease	not significant
Napster	Gnutella	B ($p < .001$)
DHT	Gnutella	A ($p < .020$)
DNS lookup	DHT lookup	B ($p < .007$)

Table 1: Survey results on comparing networked systems complexity. For each question, we present which algorithm was statistically rated as more complex based on the *t* test’s *p*-value, which indicates the probability that the result is coincidental. The smaller the *p*-value, the more significant the result.

repeated multicast, respectively. Napster was perceived as simpler than Gnutella and systems such as Gnutella and the Domain Name System (DNS) as simpler than distributed hash tables (DHTs).

The rationales for these rankings shed more insight. Participants found a system was complex if it was hard to “get right,” understand, or debug, or if it could not easily cope with failures. For the most part, issues of scalability or performance did not figure in their responses. Some sample answers include: “components have complex interactions,” “centralized or hierarchical is simpler than decentralized,” “structure is complex,” and “requires complex failure and partition handling.” Tellingly, participants at times could not clearly articulate why one algorithm was more complex than the other and resorted to *circular* definitions – *e.g.*, “chose system A because it is more complicated” or “B’s protocol is more complex.”

3 Components of Complexity

A complexity metric could make these arguments objective. A good metric would be based on quantifiable, concrete measurements of the system properties that induce implementation difficulties, complex interactions and failures, and so forth. Many metrics are possible. A perfect metric would be intuitive and easy to calculate, and would correlate with other, more subjective metrics, such as lines of code or system designers’ experience.

We build on the observation that much of system design centers on issues of state – the required state must be defined and operations for constructing and using it must be developed – but in distributed systems, one state can derive from states stored on other nodes. To calculate its state, a node must hear from the remote nodes that store the dependencies. This adds additional dependencies on the network and intermediate node states required to re-

lay input states to the node in question. Thus, not only are a given piece of state's dependencies distributed, there are also more of them.

We conjecture that the complexity particular to networked systems arises from the need to ensure state is kept in sync with its distributed dependencies. The metric we develop in this paper reflects this viewpoint and we illustrate several systems for which this dependency-centric approach appears to appropriately reflect system complexity. Alternate approaches are certainly possible however – e.g., based on protocol state machine descriptions, a protocol's state space, and so forth – and we leave a comprehensive exploration of the design space for metrics and their applicability to future work.

Our goal then is to derive a per-state measure c_s that captures the complexity due to the distributed state on which a state s depends. While a natural option would be simply to count s 's dependencies, this is not sufficiently discriminating: dependencies, like state, can vary greatly in the burden they impose. Consider Figure 1, which shows dependency relationships between states for several simple networks. In Fig. 1b, a simple distribution tree, v is computed from three dependencies w , x , and y , while in Fig. 1c, which transforms a value over several hops, v (e.g., the distance from node 1 to node 7) has just one *direct* dependency w , which is computed from $\underline{a_1}$ and x , which is in turn computed from y and $\underline{a_0}$. However, a change in y in Fig. 1b will affect *only* v , while the same change in Fig. 1c must propagate through x and w first. As a system, we argue that Fig. 1c is more complex than Fig. 1b. We therefore *weight* each state, and instead of naively counting dependencies, calculate a state's complexity by *summing* the complexities of its dependencies. The sum includes not only direct dependencies on values, but also dependencies on the *transport states* required to relay those values, accounting for networks whose transport relationships are expensive to maintain.

Some flexibility is required to account for the different types of dependencies in real networked systems, including redundancy, soft state, and so forth, and to differently penalize transport and value dependencies. Nevertheless, our metric is defined exclusively by *counting*; we avoid incorporating intricate probabilistic models of node or link behavior or state machine descriptions and the like. This keeps our metric usable, lending it to evaluation through simple examination and analysis or even empirical simulation, and represents one particular trade-off between a metric's discriminative power and the simplicity of the metric itself. Some of the limitations of our counting-based approach are discussed in Section 6.

4 A Complexity Metric

Given a system that consists of a set of states S , our goal is to assign a complexity metric c_s to each state $s \in S$.

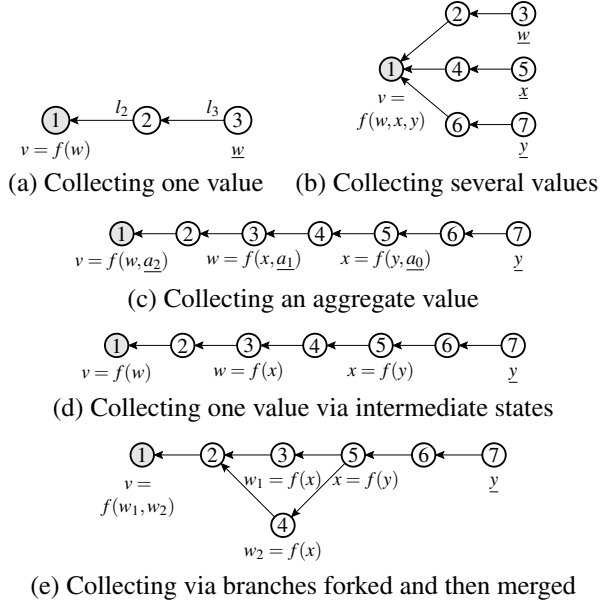


Figure 1: State relationships in four toy scenarios. For clarity routing table state, written l , is only shown in scenario (a).

We write states as lowercase letters, such as s , v , and w . Where the context is clear, we abuse notation and merge the identities of states and nodes; e.g., instead of “delivered to node 1, which stores state x ,” we simply say “delivered to x .” **Local** or **primitive** state can be maintained without network traffic, as in a sensor node's temperature reading. We sometimes indicate primitive state with an underline, as in \underline{w} . All other state is **derived** at least partially from states held at other nodes. We call these remote states **direct value dependencies**. In Fig. 1a, w is a primitive state, and v is a derived state with one value dependency, namely w , as indicated by the definition $v = f(w)$. Primitive state is assigned zero complexity, while any derived state has positive complexity. The set of state s 's direct value dependencies is written D_s .

A derived state also depends on the transport state required to relay value dependencies through the network. For instance, in Fig. 1a, propagating w to v uses the l_2 and l_3 routing table entries at nodes 2 and 3, respectively. We call these states **transport dependencies** and account for their complexity. The set $T_{s \leftarrow x}$ is defined as the set of transport dependencies involved in relaying x 's value to s ; it is empty when $x \notin D_s$. In terms of maintaining state consistency, transport dependencies are less of a burden than value dependencies since changes in a state's transport dependencies do not induce costs to keep that state in sync and, as we shall see, our metric reflects this. For instance, in Fig. 1a, any change in w must be communicated to node 1, but a change in l_2 need not, since v depends on the l states only for the delivery of w .

While some value dependencies require state changes be relayed, others need only be established once. For ex-

ample, if v were defined as a function of w at some specific time, rather than of w 's current value, then once established v is unaffected by changes elsewhere in the network. We say that state x and one of its value dependencies y are **linked** if a change in y must be propagated to x , and **unlinked** otherwise. Linked value dependencies are the major source of network complexity due to the state maintenance they incur and are treated accordingly by our metric.

Evaluating the metric requires determining dependencies among states and defining which dependencies are linked or unlinked. Unused or redundant dependencies, which frequently occur, can be measured in several ways. For example, consider Fig. 1b, where $v = f(w, x, y)$ and let us assume that the value v takes at any point is based on just one of its inputs (for instance, perhaps the active input is chosen based on minimum path length). Then v 's value dependencies have distinctly different importance: ensuring consistency requires that v and its active input stay synchronized, while updates from the other dependencies are less critical. When we consider dependencies of state v , we focus on these active states that derive v and ignore unused value dependencies.

We now turn to the metric itself, first defining a sub-metric u_s which we call the **value dependency impact**. u_s measures the number of remote states on which s is value dependent directly or indirectly. Stated otherwise, these are primitive states that, if they were to change, could result in an update at s and hence one can intuitively view u_s as indicative of the number of updates seen at s for maintaining consistency with its value dependencies. u_s is defined mutually recursively with $u_{s \leftarrow x}$, which measures the number of states on which s is value dependent via some direct value dependency $x \in D_s$. For local state s , we have $D_s = \emptyset$, $u_s = u_{s \leftarrow x} = 0$.

$$u_s = \sum_{x \in D_s} u_{s \leftarrow x};$$

$$u_{s \leftarrow x} = \begin{cases} u_x & \text{if } x \text{ is linked to } s \text{ and} \\ & x \text{ is not dependent on local state,} \\ u_x + 1 & \text{if } x \text{ is linked to } s \text{ and} \\ & x \text{ is dependent on local state,} \\ \epsilon & \text{if } x \text{ is unlinked to } s. \end{cases}$$

If the dependency $s \leftarrow x$ is linked, s must be notified of any change in $x \in D_s$. Applied recursively, changes in any of x 's direct or indirect value dependencies must also be passed on to s . Thus, the number of dependencies inherited via x is x 's own value dependency impact, u_x , plus one in the event that x was derived (in part) from local state (since a change caused by state local to x would not be accounted for in u_x). For example, states w and x in Fig. 1d, do not include any local inputs while the same states in Fig. 1c do. If s is unlinked to x , then any changes

in x are not propagated to s , so we cut off x 's value dependency impact. However, to ensure that s is charged for its initial reliance on x , we introduce ϵ , $0 < \epsilon \ll 1$, and charge this amount for every non-local, unlinked dependency.

Note that our definition of u_s assumes the dependencies s inherits are independent – a simplifying assumption due to which u_s overcounts in some dependency structures. For example, in Fig. 1e, if $v \leftarrow \{w_1, w_2\} \leftarrow x \leftarrow y$, then y is counted twice in u_v , once via w_1 and once via w_2 . This situation arose rarely. Many such branching dependency structures represent unused or redundant dependencies that we model by picking one active input, which leaves the dependency graph in the form of a tree.

The **complexity** of s is then defined as follows:

$$c_s = \sum_{x \in D_s} c_{s \leftarrow x};$$

$$c_{s \leftarrow x} = \begin{cases} u_{s \leftarrow x} + \sum_{y \in T_{s \leftarrow x}} \max(c_y, \epsilon) + c_x & \text{if } x \text{ linked,} \\ \epsilon & \text{if } x \text{ unlinked.} \end{cases}$$

This definition accounts for the entire scaffolding of distributed dependencies that maintain changes from s 's dependencies to s itself. Suppose s 's direct value dependencies are all linked. Then, the first term $\sum_{x \in D_s} u_{s \leftarrow x}$ ($= u_s$) is s 's value dependency impact. The second term $\sum_{x \in D_s} \sum_{y \in T_{s \leftarrow x}} \max(c_y, \epsilon)$ accounts for the complexity of the transport states from s 's direct value dependencies to s itself; ϵ again ensures that all links are counted, here including transport links that nominally require no state (such as one-hop wireless broadcast). Finally, the last term $\sum_{x \in D_s} c_x$ covers the complexity from inherited (transport and value) dependencies downstream from x . Local state s has $c_s = 0$. Thus intuitively, where u_s was indicative of the updates seen at s , c_s is indicative of the updates seen across all states – value and transport – that maintain changes from s 's dependencies to s .

For a chain of linked dependencies from x_0 to x_i (Fig. 1c), which depends on its local state a_i (perhaps x_i measures node i 's hop count to node 0), and writing c_i for c_{x_i} and so forth, we have $u_i = i$ and

$$c_i = i + \sum_{y \in T_{i \leftarrow (i-1)}} c_y + c_{i-1}.$$

Ignoring transport dependencies, the result is $c_i = (i^2 + i)/2$: chained linked dependencies induce complexity proportional to the square of the length of the chain. In Figure 1, if we assume all l states have complexity t , the metric yields $c_v = 1 + 2t$ in Fig. 1a, $c_v = 3 + 6t$ in Fig. 1b, and $c_v = 6 + 6t$ in Fig. 1c.

We sometimes convey intuition about the sources of complexity by writing $c_s = c_s^V + c_s^T$, where c_s^V is the complexity contributed by value dependencies and c_s^T is the

complexity contributed by transport dependencies:

$$c_{s \leftarrow x}^V = u_{s \leftarrow x} + c_x^V,$$

$$c_{s \leftarrow x}^T = \sum_{y \in T_{s \leftarrow x}} \max(c_y, \varepsilon) + c_x^T.$$

This split is purely for illustration and does not affect the definition of complexity in any way.

To measure the complexity of an *operation*, such as name resolution, routing, agreement, replication, and so forth, we simply measure the complexity of a state created or updated by that operation. For example, to measure the complexity of multihop routing, we imagine a piece of state, s , derived from one primitive value dependency, x , whose value must be routed across a multihop network. The complexity of routing is defined as c_s , which accounts for the multihop transport dependencies used to route x across the network. Assuming a network with diameter d where every routing table entry has complexity c_r , the resulting complexity is $O(dc_r)$.

This paper evaluates different networked system designs by comparing their complexities for specific operations of interest (e.g., `route`, `write_object`, `find_object`). We found it sufficient to consider one operation at a time for our evaluation. If desired, one might (for example) select the average complexity of key operations as the overall complexity of the networked system. We proceed to evaluating the above metric and defer a discussion of its scope and limitations to Section 6.

4.1 Some Canonical Scenarios

We first examine how the above complexity metric fares in evaluating a few simplified network scenarios and in the following section explore a suite of more complete networked system solutions. Before this, we first introduce two conditions that appear repeatedly in our analysis of system designs and are hence worth calling out.

Redundant inputs and paths Many systems build in redundancy to achieve higher robustness. In our analysis, this manifests itself as some state s that has multiple inputs or paths but only a subset of them are needed to derive s (akin to our discussion of active value dependencies in the previous section). For example, a multiple input scenario could be a node trying to discover the address of a wireless access point (AP) – the node listens for AP beacons but need only hear from a single AP to establish connectivity state. An example involving redundant paths might include two data centers that provision multiple disjoint network paths between them. When a message is encoded with (m, k) erasure code and each code is sent to a distinct path, the destination can construct the message if any k out of m paths work correctly. We call this the k -of- m scenario where m is the

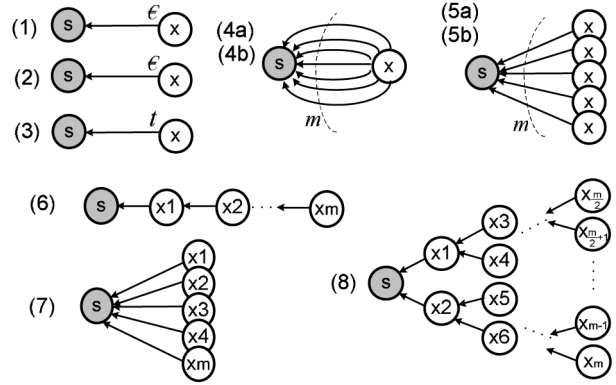


Figure 2: Canonical scenarios. For clarity we do not show the local and transport state at each node. In all scenarios other than (1) and (2), the transport state is assumed to have complexity t .

total number of inputs (paths) available and k is the number of inputs (paths) required.

We define the complexity of s derived from k -of- m inputs as being k times the average complexity due to a single input. As shown in [9], this average can be computed simply as $1/m$ times the complexity of s assuming all m inputs were required inputs.

Likewise, for the multipath scenario in which a single input x can be relayed to s using any k of m available paths, we calculate the complexities due to the transport states between x and s as k times the average complexity due to the transport states along any one path.

Recursion In some systems, a piece of state s is derived by an operation that uses states that were themselves set up by the same operation. For example, in DHTs, a node discovers its routing table entries using a `lookup` operation that makes use of state (at other nodes) that was itself set up using `lookup` operations.

We use *two complexity computation passes* for state that involves this kind of recursion. In the first pass, we compute the complexities from value and transport dependencies with the assumption that states used by the operation do not depend on the operation. We compute the final complexities in the second pass in terms of an operation on states whose complexities are computed in the first pass. In Section 5, DHTs and Paxos are canonical examples that involve recursion.

Canonical scenarios We recap the following canonical scenarios, also depicted in Figure 2. In many cases, we can construct dependency structures of networked system algorithms by composing several canonical scenarios. In all scenarios other than (1) and (2), we assume that the transport state at each node has complexity t .

(1) single input, 1-hop broadcast: here s is derived by listening to the broadcast of x . We assume x is local state and hence $c_x = 0$. Moreover, the complexity of transport state at x equals zero since broadcasting does not require any non-local transport state to be es-

Scenario	c_s
(1) 1 input, 1-hop broadcast	$1 + \epsilon$
(2) 1 non-value-dependent input, 1-hop broadcast	ϵ
(3) 1 input, 1-hop unicast	$1 + t$
(4a) 1 input, 1-of- m paths	$1 + t$
(4b) 1 input, k -of- m paths	$1 + kt$
(5a) 1-of- m inputs, 1 path	$1 + t$
(5b) k -of- m inputs, 1 path	$k(1 + t)$
(6) m inputs, in series	$\frac{1}{2}m(m+1) + mt$
(7) m inputs, in parallel	$m + mt$
(8) tree	$O(m \log m + mt)$

Table 2: Complexity of canonical scenarios

published at x . Correspondingly, state s has complexity $c_s = 1 + c_x + \max(0, \epsilon) = 1 + \epsilon$.

(2) single unlinked input, 1-hop broadcast: this case is identical to the previous case but here s is unlinked to x (e.g., s stores the value of x as soft-state) and hence $u_{s \leftarrow x} = \epsilon$ and $c_s = \epsilon$.

(3) single input, 1-hop unicast: this is identical to the first case, except that instead of broadcasting, x is routed to s using transport state at x which has complexity t and hence $c_s = 1 + t$.

(4a) single input, 1-of- m paths: this is identical to the previous case but we now have m identical paths from x to s . As before, the complexity of the transport state for each path is t and hence $c_s = 1 + t$.

(4b) single input, k -of- m paths: this is identical to the previous case but here x must be delivered to s along k paths and hence $c_s = 1 + kt$.

(5a) 1-of- m inputs, single path per input: 1 input must be delivered to s and hence $c_s = 1 + t$.

(5b) k -of- m inputs, single path per input: similar to the previous case but here k inputs must be delivered to s and hence $c_s = k(1 + t)$.

(6) m value dependencies; 1 direct, $m-1$ indirect: similar to Fig. 1c, here the value of each x_i is computed from that of x_{i+1} and local state and hence $u_s = m$ and $c_s = \frac{m(m+1)}{2} + mt$.

(7) m direct value dependencies: similar to Fig. 1b, s is computed from m inputs each of which is directly connected to s and hence $c_s = m(1 + t)$.

(8) tree: each intermediate node has two children and the tree height is $O(\log m)$. Hence $c_s = O(m \log m + mt)$.

The complexities for the above scenarios are summarized in Table 2. Comparing the complexity of s in case (6) to that in case (7), we see that dependencies that accumulate indirectly result in a higher complexity than dependencies that accumulate directly (in keeping with our discussion comparing Fig. 1b and Fig. 1c). A second observation, based on comparing cases (3) vs. (4a) or (3) vs. (5a), is that our metric neither penalizes nor

rewards the use of redundant state. This decision might seem to warrant discussion. One might argue that redundancy should add to complexity because of the additional effort that goes into creating redundant state. For example, consider a server that must create m replicas of an immutable file instead of just one. While this is true, we note that (in this example) the replicas are not dependent on each other and likewise state derived from one of the replicas is ultimately only dependent on one rather than m replicas and hence neither should have a complexity higher than if there were only a single replica. That said, the additional effort due to creating redundancy would emerge in the complexity of the operation that creates the m copies since this requires maintaining additional state to identify the m nodes at which to store replicas.

In terms of not rewarding redundancy, one might argue (as was done in [34]) that a scenario in which s is derived from k -of- m inputs should have lower complexity than if s were derived from exactly k inputs because having alternate options reduces the extent to which s depends on any single input (and similarly for paths). However, to do so would be conflating robustness and complexity¹ in the sense that having alternate inputs does not ultimately change the number of dependencies for s even though it changes the *extent* to which s might depend on any individual input; i.e., the value of s derived from k -of- m inputs does ultimately depend on some k input states.

5 Analysis

In this section we evaluate a number of networked system designs through the lens of the complexity metric defined in Section 4. Our goal in this is to: (1) illustrate the application of our metric to a broad range of systems and (2) provide concrete examples of the assessments our metric arrives at both in comparing across systems, and relative to traditional metrics.

To the extent possible, our hope is also to validate that our metric matches common design intuition. That said, conclusively validating the goodness of a metric is almost by definition difficult and, in this sense, our results are perhaps better viewed as providing the initial dataset for the future scrutiny of metric performance.

We analyzed the complexity of solutions to four problems that figure prominently in the literature on networked systems: (1) Internet routing, (2) classical distributed systems, (3) resource discovery, and (4) routing in wireless networks. Due to space constraints we only discuss the first two items in this paper; our complete set of results are presented in [9].

5.1 Routing

Routing is one of the fundamental tasks of a networked system and the literature abounds in discussions of routing architectures and algorithms. In this section we an-

analyze a set of routing solutions that represent a range of design options in terms of architecture (*e.g.*, centralized vs. distributed), scalability (*e.g.*, small vs. large tables), adoption and so forth.

For each solution, we present the complexity of an individual routing entry and a source-to-destination routing operation. For clarity we summarize only the final complexity results here and present the details of their derivation in [9]. For comparison across metrics, we also evaluate each solution using the following traditional measures: (1) per-node state, (2) number of messages and (3) convergence time.² In what follows, we consider each routing solution in turn, briefly revise its operation and summarize its complexity. The results of our analysis are summarized in Tables 3 and 4 and we end this section with a discussion examining these results.

Distance-Vector (DV) Used by protocols such as RIP and IGP, distance-vector represents one of the two major classes of IP routing solutions. DV protocols use the Bellman-Ford algorithm to calculate the shortest path between pairs of nodes. Every node maintains an estimate of its shortest distance (and corresponding next-hop) to every destination. Initially, a node is configured with the distance to its immediate neighbors and assumes a distance of infinity for all non-neighbor destinations. Each node then periodically informs its neighbors of its currently estimated distance to all destinations. For each destination, a node picks the neighbor advertising the shortest path to the destination and updates its estimated shortest distance and next-hop accordingly.

For an n node network with diameter d , DV thus requires $O(n)$ per-node state, a total message cost of $O(n^2)$ and convergence time of $O(d)$ in the absence of topology changes. In terms of our complexity measure, a single DV routing entry s has complexity $c_s = O(d^2 + d\epsilon)$ while a routing operation has a complexity of $c_{route} = O(d^3 + d^2\epsilon)$.³

Link-State (LS) Link-State routing, used in protocols such as OSPF and IS-IS, represents the second major class of widely-deployed IP routing solutions. In LS, each node floods a “link state announcement (LSA)” describing its immediate neighbor connections to the entire network. This allows each node to reconstruct the complete network topology. To compute routes, a node then simply runs Dijkstra’s algorithm over this topology map.

LS thus requires $O(nf)$ state per node (where f denotes the average node degree), incurs a total message cost of $O(n^3)$ and convergence time $O(d)$. A routing entry s has complexity $c_s = O(d + d^2\epsilon)$ while a routing operation has complexity $c_{route} = O(d^2 + d^3\epsilon)$.⁴

Centralized Architectures The authors of the 4D project [15] argue for architectures that centralize the routing control plane to simplify network management. Several subsequent proposals – RCP [5], SANE/Ethane

[7, 8], FCP [25] – present different instantiations of this centralized approach. We analyze two variants of centralized routing solutions inspired by these proposals. Our variants are not identical to any particular proposal but instead adapt their key (routing) insights for a generic network context. We do this because many of the above proposals were targeted at specific contexts which complicates drawing comparisons across solutions if we were to adopt them unchanged. For example, RCP assumes existing intra-domain routing and leverages this to deliver forwarding state from the center to the domain’s IGP routers.

In our first “RCP-inspired” variant, a designated center node collects the LSAs flooded by all nodes, reconstructs the complete network map from these LSAs, computes forwarding tables for all nodes and then uses source routing to send each node its forwarding table.⁵ When the network topology changes, the center receives the new LSA, recomputes routes and updates the forwarding state at relevant nodes. RCP-inspired has a per-state complexity of $c_s = O(d + d^2\epsilon)$ and correspondingly, a routing operation complexity of $c_{route} = O(d^2 + d^3\epsilon)$. This can be intuitively inferred by noting that a routing entry r computed at the center is similar to that at a node in LS; r is then delivered to a node in the network using a source route with the same complexity as r . RCP’s performance with traditional metrics is summarized in Table 4.

RCP-inspired centralizes the computation of routes but packet forwarding (*i.e.*, the data plane) still relies on state distributed across nodes along the path. Borrowing from several recent routing proposals [8, 25], our second variant “RCP-inspired + SR” uses source routing to forward packets between pairs of nodes. Routing construction proceeds as before but now the forwarding table sent from the center to a node A contains the entire route (as opposed to just the next hop) from A to each destination and this information is used to source route packets originating at A . Thus, rather than requiring $O(d)$ routing entries (one at each node along the path) for packet forwarding, our second variant requires only the single source-route entry at the source thus retaining the per-state complexity $c_s = O(d + d^2\epsilon)$ but lowering the complexity of c_{route} to that of a single routing entry and hence $c_{route} = O(d + d^2\epsilon)$.

Compact routing Compact routing [2, 3, 10, 36] has significantly improved scalability (*i.e.*, small routing tables) relative to deployed solutions but has seen little real-world adoption. Here, we analyze the complexity of a state-of-the-art name-independent⁶ routing algorithm by Abraham *et al.* (AG+_compact) [2]. AG+_compact guarantees optimally small routing tables of $O(\sqrt{n})$ entries, worst-case stretch less than 3.0 for arbitrary topologies and ≈ 1.0 for Internet topologies [23] and hence – as per standard measures – AG+_compact would appear

Algorithm	u_s	c_s^V	c_s^T	c_s	c_{route}
DV	$O(d)$	$O(d^2)$	$O(d\varepsilon)$	$O(d^2 + d\varepsilon)$	$O(d^3 + d^2\varepsilon)$
LS	$O(d)$	$O(d)$	$O(d^2\varepsilon)$	$O(d + d^2\varepsilon)$	$O(d^2 + d^3\varepsilon)$
RCP-inspired	$O(d)$	$O(d)$	$O(d + d^2\varepsilon)$	$O(d + d^2\varepsilon)$	$O(d^2 + d^3\varepsilon)$
RCP-inspired+SR	$O(d)$	$O(d)$	$O(d + d^2\varepsilon)$	$O(d + d^2\varepsilon)$	$O(d + d^2\varepsilon)$
Compact	$O(d\sqrt{n})$	$O(nd^2)$	$O(nd^2)$	$O(nd^2)$	$O(nd^2)$
Hierarchical LS	$O(\log \frac{n}{k})$	$O(\log \frac{n}{k})$	$O(\varepsilon \log^2 \frac{n}{k})$	$O(\log \frac{n}{k} + \varepsilon \log^2 \frac{n}{k})$	$O(\log^2 \frac{n}{k} + \varepsilon \log^3 \frac{n}{k})$
Intradomain ROFL	$O(d^2)$	$O(d^2 \log n)$	$O(d^3 \varepsilon \log n)$	$O((d^2 + d^3 \varepsilon) \log n)$	$O((d^2 + d^3 \varepsilon) \log^2 n)$

Table 3: Complexity analysis for routing solutions with the breakdown of the final per-state complexity c_s into its constituent components: u_s , the complexity contributed by value dependencies (c_s^V) and the complexity contributed by transport dependencies (c_s^T).

Algorithm	State	Message	Convergence time	Complexity
DV	$O(n)$	$O(n^2)$	$O(d)$	$O(d^3 + d^2\varepsilon)$
LS	$O(nf)$	$O(n^3)$	$O(d)$	$O(d^2 + d^3\varepsilon)$
RCP-inspired	$O(n)$, center $O(nf)$	$O(n^3)$	$O(d)$	$O(d^2 + d^3\varepsilon)$
RCP-inspired+SR	$O(n)$, center $O(nf)$	$O(n^3)$	$O(d)$	$O(d + d^2\varepsilon)$
Compact	$O(\sqrt{n})$	$O(n\sqrt{n})$	$O(d)$	$O(nd^2)$
Hierarchical LS	$O(\frac{n}{k} + k)$	$O((\frac{n}{k})^3 + k^3)$	$O(\log \frac{n}{k})$	$O(\log^2 \frac{n}{k} + \varepsilon \log^3 \frac{n}{k})$
Intradomain ROFL	$O(\log n)$	$O(n \log^2 n)$	$O(d \log^2 n)$	$O((d^2 + d^3 \varepsilon) \log^2 n)$

Table 4: Evaluation of routing solutions using different metrics

to be an attractive option for IP routing.

Briefly, `AG+_compact` operates as follows: a node A's vicinity ball (denoted $VB(A)$) is defined as the k nodes closest to A. Node A maintains routing state for every node in its own vicinity ball as well as for every node B such that $A \in VB(B)$. A distributed coloring scheme assigns every node one of c colors. One color, say red, serves as the global backbone and every node in the network maintains routing state for all red nodes. Finally, a node must know how to route to every other node of the same color as itself. For n nodes, vicinity balls of size $k = \tilde{O}(\sqrt{n})$ and $c = O(\sqrt{n})$ colors, one can show that a node's vicinity ball contains every color. With this construction, a node can always forward to a destination that is either in its own vicinity, is red, or is of the same color as the node itself. If none of these is true, the node forwards the packet to a node in its vicinity that is the same color as the destination. The challenge in `AG+_compact` lies in setting up routes between nodes of the same color without requiring state at intermediate nodes of a different color and yet maintaining bounded stretch for all paths. Loosely, `AG+_compact` achieves this as follows: say nodes A and D share the same color and A is looking to construct a routing entry to D. A explores every vicinity ball to which it belongs ($VB(I)$, $A \in VB(I)$) and that touches or overlaps the vicinity ball of the destination D (i.e., \exists node $X \in VB(I)$ with neighbor Y and $Y \in VB(D)$). For such I, A could route to D via I, X and Y. `AG+_compact` considers possible paths for each neighboring vicinity balls $VB(I)$ as well as the path through the red node closest to D and uses the shortest of these for its routing entry to D.

`AG+_compact` incurs $O(\sqrt{n})$ per-node state, total message overhead of $O(n\sqrt{n})$ and converges in $O(d)$ rounds. Derived in [9], `AG+_compact` has per-state complexity $c_s = O(nd^2)$ and $c_{route} = O(nd^2)$.

Hierarchical routing Compact routing represents one effort to reduce routing table size. The approach adopted by IP routing however has been to address scalability through the use of hierarchy. For example, OSPF may partition nodes into OSPF areas and border routers of areas are connected into a backbone network. Identifiers of nodes within a region are assigned to be aggregatable (i.e., sharing a common prefix) so that border routers need only advertise a single prefix to represent all nodes within the region.

For a network partitioned into k areas, hierarchical routing reduces the per-node state to $O(\frac{n}{k} + k)$ and total message overhead to $O((\frac{n}{k})^3 + k^3)$. The resultant complexity depends on the network topology. If the diameter of an area scales as $\log \frac{n}{k}$, then, from the LS complexity analysis, we know that routing complexity in an area is $c_a = \log^2 \frac{n}{k} + \varepsilon \log^3 \frac{n}{k}$. The final routing complexity is $2c_a$, which is asymptotically equivalent to the complexity of non-hierarchical routing $O(\log^2 n + \varepsilon \log^3 n)$. Thus, in this case, hierarchy offers improved scalability at no additional complexity. (If the network is planar, hierarchy as above actually reduces complexity by $O(\sqrt{n})$ [9].)

Intradomain ROFL Hierarchical routing offers improved scalability at the cost of constraining address assignment (giving rise to several well-documented issues). Intradomain ROFL [6] is a scalable routing protocol that retains the ability to route on flat (as opposed to aggregatable) identifiers. Each virtual node maintains

its predecessor and successor and a pointer cache that stores source routes of virtual nodes extracted from forwarded packets. In routing a packet, if a node knows a virtual node whose identifier matches the label, it sends the packet directly to the node; otherwise, it forwards the packet to a node whose identifier is closest to the label using a source route. Each node computes source routes of its neighbors from a network topology map obtained from LSAs. To simplify our analysis and comparison, we assume that the pointer cache of a node contains fingers as in Chord [35] to guarantee $O(\log n)$ hops in the flat label space and each node hosts a single virtual node representing itself.

In intradomain ROFL, a node maintains routing entries, each of which is (id, s, r) where id is a particular identifier, s is the successor of id and r is a source route to the node hosting s . Like in LS, $c_r = O(d^2 + d^3 \epsilon)$. Finding s using a lookup operation takes $O(\log n)$ hops thus yielding a complexity of $c_s = O(\log n(d^2 + d^3 \epsilon))$. A routing operation involves $\log n$ such entries, hence results in a complexity of $c_{route} = O(\log^2 n(d^2 + d^3 \epsilon))$. In other metrics, intradomain ROFL requires $O(\log n)$ state per node, incurs a total message cost of $O(n \log^2 n)$, and has convergence time $O(d \log^2 n)$.

5.1.1 Discussion

Tables 3 and 4 summarize our results which we now briefly examine. In drawing comparisons, we generally assume that the network diameter d is $O(\log n)$ and $\epsilon \sim 0$.

Complexity vs. traditional metrics Our first observation is that none of the traditional metrics yield the same relative ranking of solutions as our complexity metric, confirming that complexity (as defined here) is not the same as scalability or efficiency. Moreover, the ranking due to our complexity metric is in fair agreement with that suggested by real-world adoption and our survey results. For example, DV, LS and hierarchical routing are simpler than either AG+compact's compact routing algorithm or intradomain ROFL; centralized routing is simpler than DV, compact routing or intradomain ROFL.

Our complexity measure is also more discriminating than the other metrics. For example, DV, LS and both variants of centralized routing fare equally in terms of total state or convergence time while our metric ranks them as $DV > LS = RCP\text{-inspired} > RCP\text{-inspired} + SR$. Convergence time in particular appears too coarse-grained – for routing protocols it mostly reflects the scope to which state propagates and hence most solutions have the same value. In some sense, however, this greater discriminative power is to be expected as our metric is somewhat more complicated in the sense of taking more detail into account.

Deconstructing complexity A routing entry at a node A for destination B depends fundamentally on the link

connectivity information from the d nodes along the path to B. In DV, the computation mapping these d link states into a single routing entry is *distributed* – occurring in stages at the multiple nodes en route to A. LS by contrast, *localizes* this computation in that the d pieces of state are transferred unchanged to node A which then computes the route locally. RCP not only localizes, but *centralizes* this computation.

Our metric ranks distributed network computations as more complex than localized ones and hence DV as more complex than LS. Our metric ranks the complexity of LS as equal to that of the first centralized variant implying that a localized approach (*i.e.*, “flood everywhere then compute locally”) is similar in complexity to a centralized one (*i.e.*, “flood to a central point, compute locally, then flood from central point”). This appears justified as both approaches are ultimately similar in the number and manner in which they accumulate dependencies. While the central server can ensure an update is consistently applied in computing routes for all nodes, it is still left with the problem of consistently propagating those routes to all nodes. LS must deal with the former issue but not the latter and is thus merely making the inverse trade-off. These “simpler” approaches that localize or centralize computations might lead to greater message costs or reduced robustness and this tradeoff could be made apparent by simultaneously considering scalability, complexity and robustness metrics.

Introducing the use of source routing causes an $O(d)$ reduction in the complexity of the first RCP-inspired variant. Note too that introducing source routing to LS would result in a similar reduction. In some sense source routing localizes decision making for the *data* plane in much the same way as LS and RCP do for the control plane and hence the reduced complexity points again to the benefit of localized vs. distributed decision making. Finally, we note that, assuming $\epsilon \rightarrow 0$, the combination of LS/RCP-inspired and source routing has $O(d)$ complexity which we conjecture might be optimal for directed routing over an arbitrary topology.

In terms of navigating simplicity and scalability we note that – unlike compact routing and intradomain ROFL – introducing hierarchy improves scalability without increasing complexity.

From our analysis we find that the complexity of compact routing is in large part because of the multiple passes needed to configure routing tables – a node must first build its vicinity ball (VB), then hear from nodes whose VBs it belongs to and finally explore the intersection of “adjoining” VBs. We found a similar source of complexity in our analysis of sensornet routing algorithms (presented in [9]) that use an initial configuration phase to elect landmark nodes and then proceed to construct “virtual” coordinate systems based on distances to these

Algorithm	State	Message	Complexity
ROWAA(read)	$O(1)$	$O(1)$	$O(1)$
ROWAA(write)	$O(1)$	$O(n)$	$O(1)$
Quorum(read)	$O(1)$	$O(k)$	$O(k)$
Quorum(write)	$O(1)$	$O(k)$	$O(k^2)$
2PC	$O(1)$	$O(n)$	$O(n^2)$
Paxos	$O(1)$	$O(n)$	$O(k^3)$
Multicast	$O(n)$	$O(n)$	$O(\log^3 n)$
Gossip	$O(n)$	$O(n \log n)$	$O(\log n)$
TTL-based	1	1	ϵ
Invalidation	1	1	2

Table 5: Evaluation of classical distributed system algorithms using different metrics.

landmarks [33]. Such systems build up layers of dependencies, leading to higher complexity.

Work on compact routing is typically cast as exploring the tradeoff between efficiency (path stretch) and scalability (table size). Throwing complexity into the ring enables discussing tradeoffs between simplicity, efficiency and scalability. For example, much of the complexity of `AG+_compact` stems from the additional mechanisms needed to bound the worst-case stretch when routing between nodes in adjoining vicinities (see [9]). Were we to instead reuse the same mechanism for nodes that are in adjoining vicinity balls as for those in distant vicinities, this would reduce the complexity of `AG+_compact` to $O(\sqrt{nd}^2)$ but weaken the worst-case stretch bound.

In summary, we show that our complexity metric can discriminate across a range of routing architectures, ranks solutions in a manner that is congruent with common design intuition and can point to alternate “simpler” design options and tradeoffs.

5.2 Classical Distributed Systems

In this section, we analyze the complexity of well-known classical distributed system algorithms: (1) shared read/write variables, (2) coordination/consensus, (3) update propagation, and (4) cache consistency. For each, we consider two solutions; one that offers inferior performance/correctness guarantees relative to the other but is typically viewed as being simpler. The algorithms we analyze operate under benign fault assumptions and we assume transport states have complexity 1. We denote by n the number of servers and denote by k ($> \frac{n}{2}$) the quorum size. The results are summarized in Table 5.

5.2.1 Shared Read/Write Variable

For availability or performance, applications frequently replicate the same data on multiple servers. The replicated data can be viewed as a shared, replicated read/write variable provided by a set of servers that allow multiple clients to read from, and write to, the variable.

We compare a best-effort read-one/write-all-available (in short, ROWAA) that favors availability over consistency and quorum systems [28] used in cluster file systems such as GPFS [1]. Our analysis assumes a client knows the set of servers that participate in the algorithm.

ROWAA In ROWAA, a client issues a read request to any one of the replicas, but writes data to all available replicas in a best-effort manner. A replica that is unavailable at the time of the write is not updated and hence ROWAA can lead to inconsistency across replicas.

When a client reads a variable from a server, this fetched value (denoted by r) depends only on the current value at that server. Therefore, $c_r^V = 1$. Reading involves a request from the client to a server and the response from the server; hence $c_r^T = 2$. When a client writes a value to all available servers, it receives any acknowledgments from the servers in a best-effort manner; hence $c_w = O(1)$.

Quorum Quorum systems allow clients to tolerate some number of server faults while maintaining consistency although with lower read performance. To obtain this property, the client reads from and writes to multiple replicas, and the quorum protocol requires that there is at least one correct replica that intersects a write quorum and a read quorum thereby ensuring that the latest write is not missed by any client. For this purpose, each value stored is tagged with a timestamp.

To read a variable in a quorum system, a client sends requests to k servers and receives k (value, timestamp) pairs from a quorum. It chooses the value with the highest timestamp. Since reading a value depends on k (value, timestamp) pairs, $c^V = k$. Since there are k requests and k responses, $c^T = 2k$.

A write operation requires two phases. In the first phase, a client sends a request to read the timestamp to each of the k servers. When it receives timestamps from k servers, it chooses the value with the highest timestamp t_{high} and computes a new timestamp t_{new} greater than t_{high} . t_{new} depends on k timestamps stored at servers and these timestamps are fetched via k requests and k responses. Therefore, $c_1^V = k$ and $c_1^T = 2k$.

In the second phase, the client sends write requests (value, t_{new}) to k servers and receives acknowledgments from k servers. When a server receives this request, it updates its local state s which depends on the value and t_{new} , and hence $c_s^V = 2k + 1$ and $c_s^T = 2k + 1$. The client finishes the second phase when it receives k acknowledgments from distinct servers. Therefore, $c_2^V = k(3k + 3)$, $c_2^T = k(2k + 2)$ and hence overall complexity c is $O(k^2)$.

Observations Our complexity-based evaluation is in agreement with intuition and our survey. ROWAA has lower complexity but does not provide consistency; quorums have higher complexity but ensure consistency. This suggests that guaranteeing stronger properties (here,

consistency) may require more complex algorithms.

5.2.2 Coordination

Two-phase commit (in short, 2PC) [14] and Paxos [26] coordinate a set of servers to implement a consensus service. Both protocols operate in two phases and require a coordinator that proposes a value and a set of acceptors, which are servers that accept coordinated results. 2PC is commonly used in distributed databases and Paxos is used for replicated state machines. 2PC requires that a coordinator communicate with n servers; on the other hand, Paxos requires that a coordinator (named as a proposer in Paxos) communicate with k servers, *i.e.*, a quorum of servers (named as acceptors in Paxos). Therefore, 2PC cannot tolerate a single server fault, but Paxos can tolerate $n - k$ server faults.

2PC In the first phase of 2PC, a coordinator multicasts to R (a set of acceptors) a $\langle \text{prepare}, T \rangle$ message where T is a transaction. When an acceptor receives the message, it makes a local decision on whether to accept the transaction. If the decision is to accept T , the acceptor sends a $\langle \text{ready}, T \rangle$ message to the coordinator. Otherwise, it sends a $\langle \text{no}, T \rangle$ message to the coordinator. The coordinator collects responses from acceptors. Since the acceptor's decision depends on its local state and T sent by the coordinator, the collection at the end of the first phase has $c_1^V = 3n$. Since there are n requests sent and n responses received, the collection at the end of the first phase has $c_1^T = 2n$.

In the second phase, if the coordinator receives $\langle \text{ready}, T \rangle$ from all acceptors, it multicasts to R a $\langle \text{commit}, T \rangle$ message. Otherwise, it multicasts to R an $\langle \text{abort}, T \rangle$ message. When an acceptor receives a request for commit or abort, it executes the request and sends an $\langle \text{ack}, T \rangle$ back to the coordinator. When the coordinator receives acknowledgments from all acceptors, it knows that the transaction is completed. Since the coordinator collects n acknowledgments, $c_2^V = n(7n + 3)$ and $c_2^T = n(2n + 2)$ at the completion of the second phase. Hence 2PC has an overall complexity of $O(n^2)$.

Paxos In Paxos, each acceptor maintains two important variables: s_m that denotes the highest proposal number the acceptor promised to accept and v_a that denotes an accepted value. A proposer multicasts to R a $\langle \text{prepare}, s \rangle$ message where s is a proposal number. When an acceptor receives this message, it compares s with s_m . If $s > s_m$, the acceptor sets s_m to s and returns a $\langle \text{promise}, s, s_a, v_a \rangle$ message where s_a is the proposal number for the accepted value v_a . Otherwise, it returns an $\langle \text{error} \rangle$ message.

When the proposer receives $\langle \text{promise}, s, s_a, v_a \rangle$ messages from k distinct acceptors, it chooses v_a with the highest s_a among k messages. Let v_c and s_c be the chosen value and proposal number, respectively. If v_a is not null, v_c is set to v_a ; otherwise, v_c is set to a default value.

The proposer then multicasts to R an $\langle \text{accept}, s_c, v_c \rangle$ message. When an acceptor receives the accept message, it compares s_c with its local s_m . If $s_c \geq s_m$, s_m is set to s_c , s_a is set to s_c , and v_a is set to v_c . It then sends an $\langle \text{ack}, s_a, v_a \rangle$ message to the coordinator. Otherwise, it returns an $\langle \text{error} \rangle$ message. When the proposer receives $\langle \text{ack}, s_a, v_a \rangle$ messages from k distinct acceptors, it knows that the message is accepted by k acceptors and completes the consensus process.

Note that v_c depends on v_a 's accepted by acceptors in the second phase. To account for this dependency, we use two *passes* to compute overall complexity. In the first pass, we compute the dependency of v_a without considering the dependency in the second phase. In the second pass, we use the dependency of v_a computed in the first pass to compute the dependency in the first phase and the total dependency of the algorithm.

In the first pass, $c_{v_c}^V = 3k$ since v_c depends on k s_m 's and v_a 's, each of which depends on s sent by the proposer. Also, $c_{v_a}^V = 5k + 1$ since v_a depends on v_c and a default value. $c_{v_a}^T = 2k + 1$ since k prepare messages, k promise messages, and one accept message are required. In the second pass, $c_{v_c}^V = k(7k + 5)$ and the final $c^V = k(11k^2 + 11k + 3)$, and $c_{v_c}^T = k(2k + 3)$ and the final $c^T = k(2k^2 + 3k + 2)$. Hence Paxos has an overall complexity of $O(k^3)$.

Observations Our complexity-based evaluation is in agreement with general intuition and our survey. Both 2PC and Paxos use $O(n)$ messages, maintain $O(1)$ state per node, and have the same operation time. However, Paxos is more complex than 2PC because of interdependencies between phases. At the same time, it is this additional dependency that enables Paxos to tolerate up to $n - k$ faults while 2PC becomes unavailable with even a single fault. Our results affirm once again that guaranteeing stronger properties (here, fault-tolerance) may require more complex system algorithms.

5.2.3 Update Propagation

Update propagation algorithms disseminate an update from a publisher to all nodes (*e.g.*, publish-subscribe systems). We examine multicast (*e.g.*, ESM [20]) using a constructed tree and Gossip [11] that exchanges updates with random nodes. To ease comparison, we assume each node in the system knows k random nodes in the system from a membership service.

Multicast In multicast, nodes run DV over a k -degree mesh to build a per-source tree over which messages are disseminated. Hence forwarding state has complexity $c_s = O(\log^2 n + \epsilon \log n)$. A value received at a node depends only on the value published by the source and hence $c^V = 1$. On the other hand, if we assume the tree is balanced, $c^T = O(c_s \log n)$ and hence the overall complexity of multicast is $O(\log^3 n)$.

Gossip In Gossip, when a node receives a message, it chooses a random node and forwards the message to the selected node. This process continues until all nodes in the system receive the new update. Hence $c^V = 1$ as before. Each transport depends on a single hop from a forwarding node to a randomly chosen node, and in average $\log n$ such hops are required. Hence $c^T = O(\log n)$ and the overall complexity of Gossip is $O(\log n)$.

Observations Our metric ranks multicast as more complex than Gossip which matches our survey. However, multicast offers a deterministic guarantee of $O(\log n)$ delivery time and does so using an optimal $O(n)$ number of messages. Once again, our results convey that efficiency need not be congruent with complexity.

5.2.4 Cache Consistency

When mutable data are replicated across multiple servers, a cache consistency algorithm provides consistency across replicas. We compare TTL-based caching to invalidation-based approaches.

TTL-based caching In TTL-based caching, a cache server that receives a request first checks whether the requested data item is locally available. If so, it serves the client's request directly. Otherwise, it fetches the item from the corresponding origin server and stores the data item for its associated time-to-live (TTL). After the TTL expires, the item is evicted from the cache. Once a data item is cached, it does not depend on the item value stored at the origin server and hence a cached data item has $c = \epsilon$.

Invalidation With approaches based on invalidation, the origin server tracks which caches have copies of each data item. When a data item changes, the origin server sends an invalidation to all caches storing that item. Since a cached item depends on the master copy of the origin server, $c^V = 1$, $c^T = 1$, and $c = 2$.

Observations TTL-based caching is a soft-state technique while invalidations are a hard-state technique. Soft-state is typically viewed as simpler than hard-state because of the lack of explicit state set-up and tear-down mechanisms and our metric supports this valuation.

5.3 Other systems

Resource discovery is a fundamental problem in networked systems where information is distributed across nodes in the network. We subjected a number of well-known approaches to this problem to our complexity based analysis. Due to space constraints, because these solutions are well known in the community and our results are (we hope) fairly intuitive, we only present the final ranks of our analysis: centralized directory (*e.g.*, Napster) < (DNS, flooding-based (*e.g.*, Gnutella)) < DHT.

The derivation of the complexities and discussion of the results are described in [9].

We also analyzed several wireless routing solutions including GPSR [21] (a scalable geo routing algorithm), noGeo [33] (a scalable, but more complex solution that constructs “virtual” geographic coordinates) and AODV [31] (a less scalable but widely deployed approach). At a high level, our results (described in [9]) reflect a similar intuition as our analysis from Section 5.1 and hence we do not discuss them here.

6 Discussion

Defining a metric involves walking the line between the discriminating power of the metric (*i.e.*, the level of detail in system behavior that it can differentiate across) and the simplicity of the metric itself. Our prototype metric represents a particular point in that tradeoff. We discuss some of the implications of this choice in this section.

6.1 Limitations and possible refinements

Weighting value vs. transport dependencies Our metric assigns equal importance to value and transport dependencies. However, depending on the system environment, this may not be the best choice and a more general form of the complexity equation might be to assign:

$$c_{s \leftarrow x} = w_v u_{s \leftarrow x} + w_t \sum_{y \in T_{s \leftarrow x}} \max(c_y, \epsilon) + c_x$$

For example, a system wherein the transport state is known to be very stable while the data value of inputs change frequently might choose $w_v \gg w_t$, thus favoring system designs that incur simpler value dependencies.

Weighting dependencies Our metric treats all input or transport states as equally important. However, sometime certain input or transport states are more important (for correctness, robustness, *etc.*) than others. For example, DHTs maintain multiple routing entries but only the immediate “successor” entry ensures routing progress hence one might emphasize the complexity due to successor. Again, this might be achieved by weighting states based on system-specific knowledge of their importance.

Correlated inputs Our metric treats all inputs as independent which might result in over-counting dependencies from correlated inputs. This could be avoided by maintaining the set identifying the actual dependencies associated with each piece of state rather than just count their number although this requires significantly more fine-grained tracking of dependencies.

Capturing dependencies in time In our counting-based approach we only consider the inputs and transport states by which state was ultimately derived without worrying about the precise temporal sequence of events that led to the eventual value of state. While a time-based analysis might enable a more fine-grained view of dependencies

this would also seem more complicated since it requires incorporating a temporal model that captures the evolution of state over time.

6.2 Scope

Scalability vs. Complexity As seen in the previous sections, our complexity metric complements traditional scalability metrics. As an example of their complementary nature: our metric would not penalize system A that has the same per-state or per-operation complexity as system B but constructs more state in total than B.

Correctness vs. Complexity Our metric does little to validate the assumptions, correctness or quality of a solution. For example, our metric might capture the complexity of route construction but says little about the quality or availability of the source-to-destination path. Likewise, our metric is oblivious to undesirable assumptions that might underlie a design. For example, our metric ranks hierarchical routing favorably and cannot capture the loss in flexibility due to its requirement of aggregatable addresses (section 5.1). Similarly, our metric ranks traditional geo routing as simple despite its problematic assumption of “uniform disc” connectivity [9].

Robustness vs. Complexity Perhaps less obvious is the relationship between our complexity metric and robustness. In some sense, our metric does relate to robustness since a more complex scaffolding of dependencies does imply greater opportunities for failure. However, this relation is indirect and does not always translate to robustness. For example, consider a system where state at n nodes is derived from state at a central server. Our complexity metric would assign a low complexity to such a system, while, in terms of robustness, such a system is vulnerable to the failure of the central server.

However, we conjecture that our dependency-centric viewpoint might also apply to measuring robustness and this is something we intend to explore in future work. In particular, there are two aspects to dependencies that appear important to robustness. The first is the *vulnerability* of the system which could be captured by counting the “reverse” dependencies of a state s as the number of output states that derive from s . The second aspect is the *extent* to which a piece of state is affected by its various dependencies and this is a function of both the importance of that dependency (e.g., the address of a server vs. estimated latency to the server as a hint for better performance) and the degree to which redundancy makes the dependency less critical (i.e., deriving a piece of state from any k of m inputs with $k \ll m$ is likely more robust than one derived from k specific inputs). The former consideration (importance) can be captured by weighting dependencies as proposed above. A fairly straightforward extension to capture the effect of redundancy would be to further weight complexity by the fraction of states re-

quired; i.e., a weighted metric r_s of state s defined as: $r_s = \frac{r}{m} c_s$ where r and m are the required and available number of inputs, respectively.

7 Related Work

There is much work – particularly in software engineering – on measuring the complexity of a software *program*. For example, Halstead’s measures [18] capture programming effort derived from a program’s source code. Cyclomatic complexity [30], simply put, measures the number of decision statements. Fan in-fan out complexity [19] is a metric that measures coupling between program components as the length of code times the square of fan in times fan out. Kolmogorov complexity is measured as the length of the program’s shortest description in a description language (e.g., Turing machine). These metrics work at the level of system implementation rather than design, focus on a standalone program and do not consider the distributed dependencies of components that are networked. We believe the latter are key to capturing complexity in networked systems and both viewpoints are valuable.

Similarly, there is much work on improved approaches to system *specification* with recent efforts that focus on network contexts [22]. Metrics are complementary to system specification and cleaner specifications would make it easier to apply metrics for analysis. An interesting question for future work is whether the computation of network complexity (as we define it here) can be derived from a system specification (or even code) in an automated manner. This appears non-trivial as the *accumulation* of distributed dependencies is typically not obvious at the program or specification level.

While we derive our dependency-based metric from a system design, there have been many recent efforts at *inferring* dependencies or causality graphs from a *running* system for use in network management, troubleshooting, and performance debugging [4, 12, 16].

Finally, this paper builds on an earlier paper that articulated the need for improved complexity metrics [34].

8 Conclusions

This paper takes a first step towards quantifying the intuition for design simplicity that often guides choices for practical systems. We presented a metric that measures the impact of the ensemble of distributed dependencies for an individual piece of state and apply this metric to the evaluation of several networked system designs. While our metric is but a first step, we believe the eventual ability to more rigorously quantify design complexity would serve not only to improve our own design methodologies but also to better articulate our design aesthetic to the many communities that design for

real-world networked contexts (e.g., algorithms, formal distributed systems, graph theory).

Acknowledgments

We thank Matthew Caesar, Rodrigo Fonseca, Paul Francis, Brighton Godfrey, Robert Kleinberg, Henry Lin, Scott Shenker, David Wetherall, the anonymous reviewers and our shepherd, Ken Birman, for their comments.

References

- [1] IBM General Parallel File System (GPFS).
- [2] I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. In *Proc. of SPAA*, 2004.
- [3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive routing. In *Proc. of STOC*, 1989.
- [4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.
- [5] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. of NSDI*, 2005.
- [6] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on flat labels. In *SIGCOMM*, 2006.
- [7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.
- [8] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. of USENIX Security*, 2006.
- [9] B.-G. Chun, S. Ratnasamy, and E. Kohler. A complexity metric for networked system designs. IRB-TR-07-010.
- [10] L. J. Cowen. Compact routing with minimum stretch. In *Proc. of SODA*, 1999.
- [11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of PODC*, 1987.
- [12] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proc. of NSDI*, 2007.
- [13] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. *SigACT News*, 2002.
- [14] J. Gray. Notes on database systems. *IBM Research Report RJ2188*, Feb. 1978.
- [15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. In *SIGCOMM CCR*, 2005.
- [16] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proc. of DSOM*, 1998.
- [17] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. of OSDI*, 2004.
- [18] M. Halstead. *Elements of Software Science, Operating, and Programming Systems*. Elsevier, 1977.
- [19] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, pages 510–518, 1981.
- [20] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. of SIGMETRICS*, 2000.
- [21] B. Karp and H. T. Kung. Greedy perimeter stateless routing for wireless networks. In *Proc. of MOBICOM*, 2000.
- [22] M. Karsten, S. Keshav, S. Prasad, and O. Beg. An axiomatic basis for communication. In *SIGCOMM*, 2007.
- [23] D. Krioukov, K. Fall, and X. Yang. Compact routing on Internet-like graphs. In *Proc. of INFOCOM*, 2004.
- [24] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *Proc. of PODC*, 2006.
- [25] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, and S. Shenker. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.
- [26] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.
- [27] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, pages 193–201, 1992.
- [28] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of STOC*, 1997.
- [29] Y. Mao, F. Wang, L. Qiu, S. S. Lam, and J. M. Smith. S4: Small state and small stretch routing protocol for large wireless sensor networks. In *Proc. of NSDI*, 2007.
- [30] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Comm. of the ACM*, pages 1415–1425, 1989.
- [31] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of WMCSA*, 1999.
- [32] R. Perlman, C.-Y. Lee, A. Ballardie, J. Crowcroft, Z. Wang, T. Maufer, C. Diot, J. Thoo, and M. Green. Simple multicast: A design for simple, low-overhead multicast. Internet-Draft draft-perlman-simple-multicast-03, Internet Engineering Task Force, Oct. 1999. Work in progress.
- [33] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proc. of MOBICOM*, 2003.
- [34] S. Ratnasamy. Capturing complexity in networked systems design: The case for improved metrics. In *Proc. of HotNets*, 2006.
- [35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.
- [36] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of SPAA*, 2001.

Notes

¹We thank Paul Francis and Robert Kleinberg for discussion on this.

²We include this since the time complexity of distributed algorithms is commonly used in the theory community [24,27]. Time complexity is the maximum number of message-exchange rounds needed to complete the required computation.

³This can be inferred by noting that route construction is similar to the canonical “ m inputs in series” scenario from the previous section.

⁴This is quickly inferred by noting the similarity to the “ m inputs in parallel” scenario with $m = d$ inputs relayed along a path of $O(d)$ hops and transport state of complexity ϵ at each hop.

⁵This use of source routing is the key difference relative to RCP which uses the underlying intra-domain routes for the same purpose.

⁶We do not consider name-dependent algorithms [10,29] as these require an additional name translation service for IP routing.

DieCast: Testing Distributed Systems with an Accurate Scale Model

Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat

University of California, San Diego

{dgupta, kvishwanath, vahdat}@cs.ucsd.edu

Abstract

Large-scale network services can consist of tens of thousands of machines running thousands of unique software configurations spread across hundreds of physical networks. Testing such services for complex performance problems and configuration errors remains a difficult problem. Existing testing techniques, such as simulation or running smaller instances of a service, have limitations in predicting overall service behavior.

Although technically and economically infeasible at this time, testing should ideally be performed at the same scale and with the same configuration as the deployed service. We present DieCast, an approach to scaling network services in which we multiplex all of the nodes in a given service configuration as virtual machines (VM) spread across a much smaller number of physical machines in a test harness. CPU, network, and disk are then accurately scaled to provide the illusion that each VM matches a machine from the original service in terms of both available computing resources and communication behavior to remote service nodes. We present the architecture and evaluation of a system to support such experimentation and discuss its limitations. We show that for a variety of services—including a commercial, high-performance, cluster-based file system—and resource utilization levels, DieCast matches the behavior of the original service while using a fraction of the physical resources.

1 Introduction

Today, more and more services are being delivered by complex systems consisting of large ensembles of machines spread across multiple physical networks and geographic regions. Economies of scale, incremental scalability, and good fault isolation properties have made clusters the preferred architecture for building planetary-scale services. A single logical request may touch dozens of machines on multiple networks, all providing instances of services transparently replicated across multiple machines. Services consisting of tens of thousands of machines are commonplace [11].

Economic considerations have pushed service providers to a regime where individual service machines

must be made from commodity components—saving an extra \$500 per node in a 100,000-node service is critical. Similarly, nodes run commodity operating systems, with only moderate levels of reliability, and custom-written applications that are often rushed to production because of the pressures of “Internet Time.” In this environment, failure is common [24] and it becomes the responsibility of higher-level software architectures, usually employing custom monitoring infrastructures and significant service and data replication, to mask individual, correlated, and cascading failures from end clients.

One of the primary challenges facing designers of modern network services is testing their dynamically evolving system architecture. In addition to the sheer scale of the target systems, challenges include: heterogeneous hardware and software, dynamically changing request patterns, complex component interactions, failure conditions that only manifest under high load [21], the effects of correlated failures [20], and bottlenecks arising from complex network topologies. Before upgrading any aspect of a networked service—the load balancing/replication scheme, individual software components, the network topology—architects would ideally create an exact copy of the system, modify the single component to be upgraded, and then subject the entire system to both historical and worst-case workloads. Such testing must include subjecting the system to a variety of controlled failure and attack scenarios since problems with a particular upgrade will often only be revealed under certain specific conditions.

Creating an exact copy of a modern networked service for testing is often technically challenging and economically infeasible. The architecture of many large-scale networked services can be characterized as “controlled chaos,” where it is often impossible to know exactly what the hardware, software, and network topology of the system looks like at any given time. Even when the precise hardware, software and network configuration of the system is known, the resources to replicate the production environment might simply be unavailable, particularly for large services. And yet, reliable, low overhead, and economically feasible testing of network services remains critical to delivering robust higher-level services.

The goal of this work is to develop a testing method-

ology and architecture that can accurately predict the behavior of modern network services while employing an order of magnitude less hardware resources. For example, consider a service consisting of 10,000 heterogeneous machines, 100 switches, and hundreds of individual software configurations. We aim to configure a smaller number of machines (e.g., 100-1000 depending on service characteristics) to emulate the original configuration as closely as possible and to subject the test infrastructure to the same workload and failure conditions as the original service. The performance and failure response of the test system should closely approximate the real behavior of the target system. Of course, these goals are infeasible without giving something up: if it were possible to capture the complex behavior and overall performance of a 10,000 node system on 1,000 nodes, then the original system should likely run on 1,000 nodes.

A key insight behind our work is that we can trade *time* for system capacity while accurately scaling individual system components to match the behavior of the target infrastructure. We employ *time dilation* to accurately scale the capacity of individual systems by a configurable factor [19]. Time dilation fully encapsulates operating systems and applications such that the rate at which time passes can be modified by a constant factor. A time dilation factor (TDF) of 10 means that for every second of real time, all software in a dilated frame believes that time has advanced by only 100 ms. If we wish to subject a target system to a one-hour workload when scaling the system by a factor of 10, the test would take 10 hours of real time. For many testing environments, this is an appropriate tradeoff. Since the passage of time is slowed down while the rate of *external events* (such as network I/O) remains unchanged, the system appears to have substantially higher processing power and faster network and disk.

In this paper, we present DieCast, a complete environment for building accurate models of network services (Section 2). Critically, we run the actual operating systems and application software of some target environment on a fraction of the hardware in that environment. This work makes the following contributions. First, we extend our original implementation of time dilation [19] to support fully virtualized as well as paravirtualized hosts. To support complete system evaluations, our second contribution shows how to extend dilation to disk and CPU (Section 3). In particular, we integrate a full disk simulator into the virtual machine monitor (VMM) to consider a range of possible disk architectures. Finally, we conduct a detailed system evaluation, quantifying DieCast's accuracy for a range of services, including a commercial storage system (Sections 4 and 5). The goals of this work are ambitious and while we cannot claim to have addressed all of the myriad chal-

lenges associated with testing large-scale network services (Section 6), we believe that DieCast shows significant promise as a testing vehicle

2 System Architecture

We begin by providing an overview of our approach to scaling a system down to a target test harness. We then discuss the individual components of our architecture.

2.1 Overview

Figure 1 gives an overview of our approach. On the left (Figure 1(a)) is an abstract depiction of a network service. A load balancing switch sits in front of the service and redirects requests among a set of front-end HTTP servers. These requests may in turn travel to a middle tier of application servers, who may query a storage tier consisting of databases or network attached storage.

Figure 1(b) shows how a target service can be scaled with DieCast. We encapsulate all nodes from the original service in virtual machines and multiplex several of these VMs onto physical machines in the test harness. Critically, we employ time dilation in the VMM running on each physical machine to provide the illusion that each virtual machine has, for example, as much processing power, disk I/O, and network bandwidth as the corresponding host in the original configuration despite the fact that it is sharing underlying resources with other VMs. DieCast configures VMs to communicate through a network emulator to reproduce the characteristics of the original system topology. We then initialize the test system using the setup routines of the original system and subject it to appropriate workloads and fault-loads to evaluate system behavior.

The overall goal is to improve predictive power. That is, runs with DieCast on smaller machine configurations should accurately predict the performance and fault tolerance characteristics of some larger production system. In this manner, system developers may experiment with changes to system architecture, network topology, software upgrades, and new functionality before deploying them in production. Successful runs with DieCast should improve confidence that any changes to the target service will be successfully deployed. Below, we discuss the steps in applying our general approach to applying DieCast scaling to target systems.

2.2 Choosing the Scaling Factor

The first question to address is the desired scaling factor. One use of DieCast is to reproduce the scale of an original service in a test cluster. Another application is to scale existing test harnesses to achieve more realism than possible from the raw hardware. For instance, if 100 nodes are already available for testing, then DieCast might be employed to scale to a thousand-node system

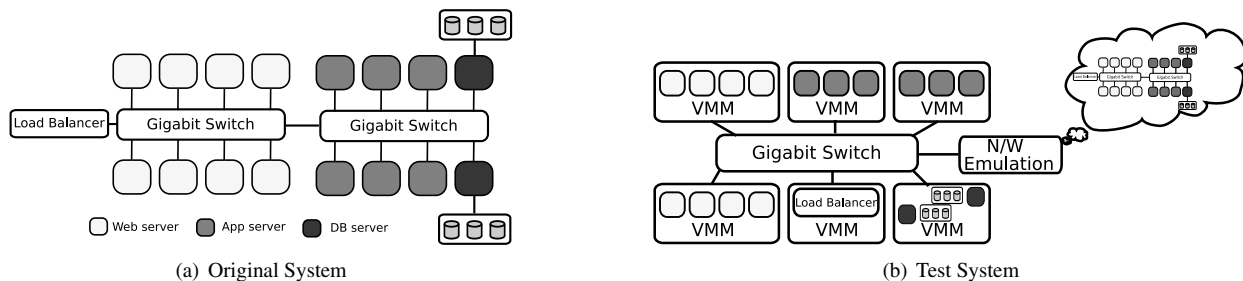


Figure 1: Scaling a network service to the DieCast infrastructure.

with a more complex communication topology. While the DieCast system may still fall short of the scale of the original service, it can provide more meaningful approximations under more intense workloads and failure conditions than might have otherwise been possible.

Overall, the goal is to pick the largest scaling factor possible while still obtaining accurate predictions from DieCast, since the prediction accuracy will naturally degrade with increasing scaling factors. This maximum scaling factor depends on the characteristics of the target system. Section 6 highlights the potential limitations of DieCast scaling. In general, scaling accuracy will degrade with: i) application sensitivity to the fine-grained timing behavior of external hardware devices; ii) capacity-constrained physical resources; and iii) system devices not amenable to virtualization. In the first category, application interaction with I/O devices may depend on the exact timing of requests and responses. Consider for instance a fine-grained parallel application that assumes all remote instances are co-scheduled. A DieCast run may mispredict performance if target nodes are not scheduled at the time of a message transmission to respond to a blocking read operation. If we could interleave at the granularity of individual instructions, then this would not be an issue. However, context switching among virtual machines means that we must pick time slices on the order of milliseconds. Second, DieCast cannot scale the capacity of hardware components such as main memory, processor caches, and disk. Finally, the original service may contain devices such as load balancing switches that are not amenable to virtualization or dilation. Even with these caveats, we have successfully applied scaling factors of 10 to a variety of services with near-perfect accuracy as discussed in Sections 4 and 5.

Of the above limitations to scaling, we consider capacity limits for main memory and disk to be most significant. However, we do not believe this to be a fundamental limitation. For example, one partial solution is to configure the test system with more memory and storage than the original system. While this will reduce some of the economic benefits of our approach, it will not erase them. For instance, doubling a machine's memory will not typically double its hardware cost. More importantly, it will

not substantially increase the typically dominant human cost of administering a given test infrastructure because the number of required administrators for a given test harness usually grows with the number of machines in the system rather than with the total memory of the system.

Looking forward, ongoing research in VMM architectures have the potential to reclaim some of the memory [32] and storage overhead [33] associated with multiplexing VMs on a single physical machine. For instance, four nearly identically configured Linux machines running the same web server will overlap significantly in terms of their memory and storage footprints. Similarly, consider an Internet service that replicates content for improved capacity and availability. When scaling the service down, multiple machines from the original configuration may be assigned to a single physical machine. A VMM capable of detecting and exploiting available redundancy could significantly reduce the incremental storage overhead of multiplexing multiple VMs.

2.3 Cataloging the Original System

The next task is to configure the appropriate virtual machine images onto our test infrastructure. Maintaining a catalog of the hardware and software configuration that comprises an Internet service is challenging in its own right. However, for the purposes of this work, we assume that such a catalog is available. This catalog would consist of all of the hardware making up the service, the network topology, and the software configuration of each node. The software configuration includes the operating system, installed packages and applications, and the initialization sequence run on each node after booting.

The original service software may or may not run on top of virtual machines. However, given the increasing benefits of employing virtual machines in data centers for service configuration and management and the popularity of VM-based appliances that are pre-configured to run particular services [7], we assume that the original service is in fact VM-based. This assumption is not critical to our approach but it also partially addresses any baseline performance differential between a node running on

bare hardware in the original service and the same node running on a virtual machine in the test system.

2.4 Configuring the Virtual Machines

With an understanding of appropriate scaling factors and a catalog of the original service configuration, DieCast then configures individual physical machines in the test system with multiple VM images reflecting, ideally, a one-to-one map between physical machines in the original system and virtual machines in the test system. With a scaling factor of 10, each physical node in the target system would host 10 virtual machines. The mapping from physical machines to virtual machines should account for: similarity in software configurations, per-VM memory and disk requirements and the capacity of the hardware in the original and test system. In general, a solver may be employed to determine a near-optimal matching [26]. However, given the VM migration capabilities of modern VMMs and DieCast's controlled network emulation environment, the actual location of a VM is not as significant as in the original system.

DieCast then configures the VMs such that each VM appears to have resources identical to a physical machine in the original system. Consider a physical machine hosting 10 VMs. DieCast would run each VM with a scaling factor of 10, but allocate each VM only 10% of the actual physical resource. DieCast employs a non-work conserving scheduler to ensure that each virtual machine receives no more than its allotted share of resources even when spare capacity is available. Suppose a CPU intensive task takes 100 seconds to finish on the original machine. The same task would now take 1000 seconds (of real time) on a dilated VM, since it can only use a tenth of the CPU. However, since the VM is running under time dilation, it only perceives that 100 seconds have passed. Thus in the VMs time frame, resources appear equivalent to the original machine. We only explicitly scale CPU and disk I/O latency on the host; scaling of network I/O happens via network emulation as described next.

2.5 Network Emulation

The final step in the configuration process is to match the network configuration of the original service using network emulation. We configure all VMs in the test system to route all their communication through our emulation environment. Note that DieCast is not tied to any particular emulation technology: we have successfully used DieCast with Dummynet [27], Modelnet [31] and Netem [3] where appropriate.

It is likely that the bisection bandwidth of the original service topology will be larger than that available in the test system. Fortunately, time dilation is of significant value here. Convincing a virtual machine scaled by a factor of 10 that it is receiving data at 1 Gbps only

requires forwarding data to it at 100 Mbps. Similarly, it may appear that latencies in an original cluster-based service may be low enough that the additional software forwarding overhead associated with the emulation environment could make it difficult to match the latencies in the original network. To our advantage, maintaining accurate latency with time dilation actually requires *increasing* the real time delay of a given packet; e.g., a 100 μ s delay network link in the original network should be delayed by 1 ms when dilating by a factor of 10.

Note that the scaling factor need not match the TDF. For example, if the original network topology is so large/fast that even with a TDF of 10 the network emulator is unable to keep up, it is possible to employ a time dilation factor of 20 while maintaining a scaling factor of 10. In such a scenario, there would still on average be 10 virtual machines multiplexed onto each physical machine, however the VMM scheduler would allocate only 5% of the physical machine's resources to individual machines (meaning that 50% of CPU resources will go idle). The TDF of 20, however, would deliver additional capacity to the network emulation infrastructure to match the characteristics of the original system.

2.6 Workload Generation

Once DieCast has prepared the test system to be *resource equivalent* to the original system, we can subject it to an appropriate workload. These workloads will in general be application-specific. For instance, Monkey [15] shows how to replay a measured TCP request stream sent to a large-scale network service. For this work, we use application-specific workload generators where available and in other cases write our own workload generators that both capture normal behavior as well as stress the service under extreme conditions.

To maintain a target scaling factor, clients should also ideally run in DieCast-scaled virtual machines. This approach has the added benefit of allowing us to subject a test service to a high level of perceived-load using relatively few resources. Thus, DieCast scales not only the capacity of the test harness but also the workload generation infrastructure.

3 Implementation

We have implemented DieCast support on several versions of Xen [10]: v2.0.7, v3.0.4, and v3.1 (both paravirtualized and fully virtualized VMs). Here we focus on the Xen 3.1 implementation. We begin with a brief overview of time dilation [19] and then describe the new features required to support DieCast.

3.1 Time Dilation

Critical to time dilation is a VMM's ability to modify the perception of time within a guest OS. Fortunately, most

VMMs already have this functionality, for example, because a guest OS may develop a backlog of “lost ticks” if it is not scheduled on the physical processor when it is due to receive a timer interrupt. Since the guest OS running in a VM does not run continuously, VMMs periodically synchronize the guest OS time with the physical machine’s clock. The only requirement for a VMM to support time dilation is this ability to modify the VM’s perception of time. In fact, as we demonstrate in Section 5, the concept of time dilation can be ported to other (non-virtualized) environments.

Operating systems employ a variety of time sources to keep track of time, including timer interrupts (e.g., the Programmable Interrupt Timer or PIT), specialized counters (e.g., the TSC on Intel platforms) and external time sources such as NTP. Time dilation works by intercepting the various time sources and scaling them appropriately to fully encapsulate the OS in its own time frame.

Our original modifications to Xen for paravirtualized hosts [19] therefore appropriately scale time values exposed to the VM by the hypervisor. Xen exposes two notions of time to VMs. Real time is the number of nanoseconds since boot, and wall clock time is the traditional Unix time since epoch. While Xen allows the guest OS to maintain and update its own notion of time via an external time source (such as NTP), the guest OS often relies solely on Xen to maintain accurate time. Real and wall clock time pass between the Xen hypervisor and the guest operating system via a shared data structure. Dilation uses a per-domain TDF variable to appropriately scale real time and wall clock time. It also scales the frequency of timer interrupts delivered to a guest OS since these timer interrupts often drive the internal time keeping of a guest. Given these modifications to Xen, our earlier work showed that network dilation matches undilated baselines for complex per-flow TCP behavior in a variety of scenarios [19].

3.2 Support for OS diversity

Our original time dilation implementation only worked with paravirtualized machines, with two major drawbacks: it supported only Linux as the guest OS, and the guest kernel required modifications. Generalizing to other platforms would have required code modifications to the respective OS. To be widely applicable, DieCast must support a variety of operating systems.

To address these limitations, we ported time dilation to support *fully virtualized* (FV) VMs, enabling DieCast to support unmodified OS images. Note that FV VMs require platforms with hardware support for virtualization, such as Intel VT or AMD SVM. While Xen support for fully virtualized VMs differs significantly from the paravirtualized VM support in several key areas such as I/O emulation, access to hardware registers, and time man-

agement, the general idea behind the implementation remains the same: we want to intercept all sources of time and scale them.

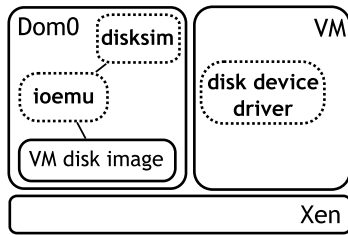
In particular, our implementation scales the PIT, the TSC register (on x86), the RTC (Real Time Clock), the ACPI power management timer and the High Performance Event Timer (HPET). As in the original implementation, we also scale the number of timer interrupts delivered to a fully virtualized guest. We allow each VM to run with an independent scaling factor. Note, however, that the scaling factor is fixed for the life time of a VM—it can not be changed at run time.

3.3 Scaling Disk I/O and CPU

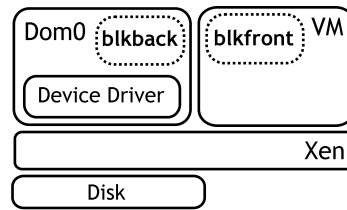
Time dilation as described in [19] did not scale disk performance, making it unsuitable for services that perform significant disk I/O. Ideally, we would scale individual disk requests at the disk controller layer. The complexity of modern drive architectures, particularly the fact that much low level functionality is implemented in firmware, makes such implementations challenging. Note that simply delaying requests in the device driver is not sufficient, since disk controllers may re-order and batch requests for efficiency. On the other hand, functionality embedded in hardware or firmware is difficult to instrument and modify. Further complicating matters are the different I/O models in Xen: one for paravirtualized (PV) VMs and one for fully virtualized (FV) VMs. DieCast provides mechanisms to scale disk I/O for both models.

For FV VMs, DieCast integrates a highly accurate and efficient disk system simulator — Disksim [17] — which gives us a good trade-off between realism and accuracy. Figure 2(a) depicts our integration of DiskSim into the fully virtualized I/O model: for each VM, a dedicated user space process (`ioemu`) in Domain-0 performs I/O emulation by exposing a “virtual disk” to the VM (the guest OS is unaware that a real disk is not present). A special file in Domain-0 serves as the backend storage for the VM’s disk. To allow `ioemu` to interact with DiskSim, we wrote a wrapper around the simulator for inter-process communication.

After servicing each request (but before returning), `ioemu` forwards the request to Disksim, which then returns the time, rt , the request would have taken in its simulated disk. Since we are effectively layering a software disk on top of `ioemu`, each request should ideally take exactly time rt in the VM’s time frame, or $tdf * rt$ in real time. If $delay$ is the amount by which this request is delayed, the total time spent in `ioemu` becomes $delay + dt + st$, where st is the time taken to *actually* serve the request (Disksim only simulates I/O characteristics, it does not deal with the actual disk content) and dt is the time taken to invoke Disksim itself. The required delay is then $(tdf * rt) - dt - st$.



(a) I/O Model for FV VMs



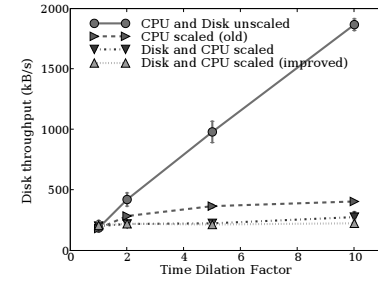
(b) I/O Model for PV VMs

Figure 2: Scaling Disk I/O

The architecture of Disksim, however, is not amenable to integration with the PV I/O model (Figure 2(b)). In this “split I/O” model, a front-end driver in the VM (*blkfront*) forwards requests to a back-end driver in Domain-0 (*blkback*), which are then serviced by the real disk device driver. Thus PV I/O is largely a kernel activity, while Disksim runs entirely in user-space. Further, a separate Disksim process would be required for each simulated disk, whereas there is a single back-end driver for all VMs.

For these reasons, for PV VMs, we inject the appropriate delays in the *blkfront* driver. This approach has the additional advantage of containing the side effects of such delays to individual VMs — *blkback* can continue processing other requests as usual. Further, it eliminates the need to modify disk-specific drivers in Domain-0. We emphasize that this is functionally equivalent to per-request scaling in Disksim: the key difference is that scaling in Disksim is much closer to the (simulated) hardware. Overall our implementation of disk scaling for PV VM’s is simpler though less accurate and somewhat less flexible since it requires the disk subsystem in the testing hardware to match the configuration in the target system.

We have validated both our implementations using several micro-benchmarks. For brevity, we only describe one of them here. We run DBench [29] — a popular hard-drive and file-system benchmark — under different dilation factors and plot the reported throughput. Figure 2(c) shows the results for the FV I/O model with Disksim integration (results for the PV implementation can be found in a separate technical report [18]). Ideally, the throughput should remain constant as a function of the dilation factor. We first run the benchmark without scaling disk I/O or CPU, and we can see that the reported throughput increases almost linearly, an undesirable behavior. Next, we repeat the experiment and scale the CPU alone (thus, at TDF 10 the VM only receives 10% of the CPU). While the increase is no longer linear, in the absence of disk dilation it is still significantly higher than the expected value. Finally, with disk dilation in place we can see that the throughput closely tracks the expected value.



(c) DBench throughput under Disksim

However, as the TDF increases, we start to see some divergence. After further investigation, we found that this deviation results from the way we scaled the CPU. Recall that we scale the CPU by bounding the amount of CPU available to each VM. Initially, we simply used Xen’s Credit scheduler to allocate an appropriate fraction of CPU resources to each VM in non-work conserving mode. However, simply scaling the CPU does not govern how those CPU cycles are distributed across time. With the original Credit scheduler, if a VM does not consume its full timeslice, it can be scheduled again in subsequent timeslices. For instance, if a VM is set to be diluted by a factor of 10 and if it consumes less than 10% of the CPU in each time slice, then it will run in *every* time slice, since in aggregate it never consumes more than its hard bound of 10% of the CPU. This potential to run continuously distorts the performance of I/O-bound applications under dilation, and in particular they’ll have a different timing distribution than they would in the real time frame. This distortion increases with increasing TDF. Thus, we found that, for some workloads, we may actually wish to enforce that the VM’s CPU consumption should be more *uniformly* enforced across time.

We modified the Credit CPU scheduler in Xen to support this mode of operation as follows: if a VM runs for the entire duration of its time slice, we ensure that it does *not* get scheduled for the next $(tdf - 1)$ time slices. If a VM voluntarily yields the CPU or is pre-empted before its time slice expires, it *may* be re-scheduled in a subsequent time slice. However, as soon as it consumes a cumulative total of a time slice’s worth of run time (carried over from the previous time it was descheduled), it will be pre-empted and not allowed to run for another $(tdf - 1)$ time slices. The final line in figure 2(c) shows the results of the DBench benchmark with using this modified scheduler. As we can see, the throughput remains consistent even at higher TDFs. Note that unlike in this benchmark, DieCast typically runs multiple VMs per machine, in which case this “spreading” of CPU cycles occurs naturally as VMs compete for CPU.

4 Evaluation

We seek to answer the following questions with respect to DieCast-scaling: i) Can we configure a smaller number of physical machines to match the CPU capacity, complex network topology, and I/O rates of a larger service? ii) How well does the performance of a scaled service running on fewer resources match the performance of a baseline service running with more resources? we consider three different systems: i) BitTorrent, a popular peer-to-peer file sharing program; ii) RUBiS, an auction service prototyped after eBay; and iii) Isaac, our configurable network three-tier service that allows us to generate a range of workload scenarios.

4.1 Methodology

To evaluate DieCast for a given system, we first establish the baseline performance: this involves determining the configuration(s) of interest, fixing the workload, and benchmarking the performance. We then scale the system down by an order of magnitude and compare the DieCast performance to the baseline. While we have extensively evaluated DieCast implementations for several versions of Xen, we only present the results for the Xen 3.1 implementation here. Detailed evaluation for Xen 3.0.4 can be found in our technical report [18].

Each physical machine in our testbed is a dual-core 2.3GHz Intel Xeon with 4GB RAM. Note that since the DiskSim integration only works with fully virtualized VMs, for a fair evaluation it is *required* that even the baseline system run on VMs—ideally the baseline would be run on physical machines directly (for the paravirtualized setup, we do have evaluation with physical machines as the baseline. We refer the reader to [18] for details). We configure DiskSim to emulate a Seagate ST3217 disk drive. For the baseline, DiskSim runs as usual (no requests are scaled) and with DieCast, we scale each request as described in Section 3.3.

We configure each virtual machine with 256MB RAM and run Debian Etch on Linux 2.6.17. Unless otherwise stated, the baseline configuration consists of 40 physical machines hosting a single VM each. We then compare the performance characteristics to runs with DieCast on four physical machines hosting 10 VMs each, scaled by a factor of 10. We use Modelnet for the network emulation, and appropriately scale the link characteristics for DieCast. For allocating CPU, we use our modified Credit CPU scheduler as described in Section 3.3.

4.2 BitTorrent

We begin by using DieCast to evaluate BitTorrent [1] — a popular P2P application. For our baseline experiments, we run BitTorrent (version 3.4.2) on a total of 40 virtual machines. We configure the machines to communicate

across a ModelNet-emulated dumbbell topology (Figure 3), with varying bandwidth and latency values for the access link (A) from each client to the dumbbell and the dumbbell link itself (C). We vary the total number of clients, the file size, the network topology, and the version of the BitTorrent software. We use the distribution of file download times across all clients as the metric for comparing performance. The aim here is to observe how closely DieCast-scaled experiments reproduce behavior of the baseline case for a variety of scenarios.

The first experiment establishes the baseline where we compare different configurations of BitTorrent sharing a file across a 10Mbps dumbbell link and constrained access links of 10Mbps. All links have a one-way latency of 5ms. We run a total of 40 clients (with half on each side of the dumbbell). Figure 5 plots the cumulative distribution of transfer times across all clients for different file sizes (10MB and 50MB). We show the baseline case using solid lines and use dashed lines to represent the DieCast-scaled case. With DieCast scaling, the distribution of download times closely matches the behavior of the original system. For instance, well-connected clients on the same side of the dumbbell as the randomly chosen seeder finish more quickly than the clients that must compete for scarce resources across the dumbbell.

Having established a reasonable baseline, we next consider sensitivity to changing system configurations. We first vary the network topology by leaving the dumbbell link unconstrained (1 Gbps) with results in Figure 5. The graph shows the effect of removing the bottleneck on the finish times compared to the constrained dumbbell-link case for the 50-MB file: all clients finish within a small time difference of each other as shown by the middle pair of curves.

Next, we consider the effect of varying the total number of clients. Using the topology from the baseline experiment we repeat the experiments for 80 and 200 simultaneous BitTorrent clients. Figure 6 shows the results. The curves for the baseline and DieCast-scaled versions almost completely overlap each other for 80 clients (left pair of curves) and show minor deviation from each other for 200 clients (right pair of curves). Note that with 200 clients, the bandwidth contention increases to the point where the dumbbell bottleneck becomes less important.

Finally, we consider an experiment that demonstrates the flexibility of DieCast to reproduce system performance under a variety of resource configurations starting with the same baseline. Figure 7 shows that in addition to matching 1:10 scaling using 4 physical machines hosting 10 VMs each, we can also match an alternate configuration of 8 physical machines, hosting five VMs each with a dilation factor of five. This demonstrates that even if it is necessary to vary the number of physical machines available for testing, it may still be possible to find

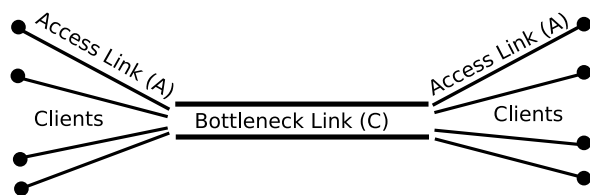


Figure 3: Topology for BitTorrent experiments.

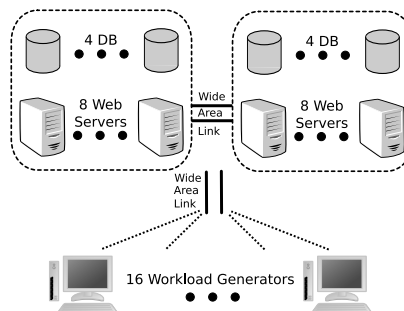


Figure 4: RUBiS Setup.

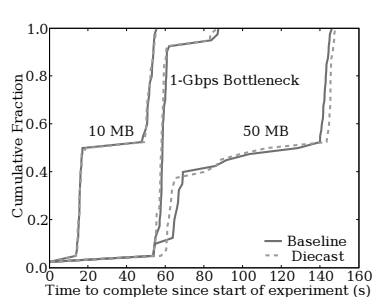


Figure 5: Performance with varying file sizes.

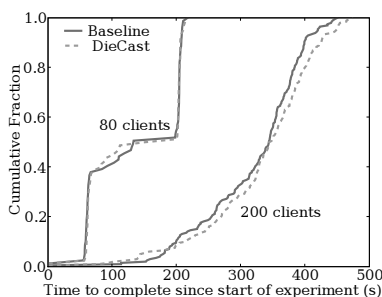


Figure 6: Varying #clients.

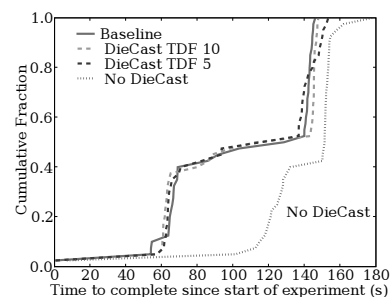


Figure 7: Different configurations.

an appropriate scaling factor to match performance characteristics. This graph also has a fourth curve, labeled “No DieCast”, corresponding to running the experiment with 40 VMs on four physical machines, each with a dilation factor of 1—disk and network are *not* scaled (thus match the baseline configuration), and all VMs are allocated equal shares of the CPU. This corresponds to the approach of simply multiplexing a number of virtual machines on physical machines without using DieCast. The graph shows that the behavior of the system under such a naive approach varies widely from actual behavior.

4.3 RUBiS

Next, we investigate DieCast’s ability to scale a fully functional Internet service. We use RUBiS [6]—an auction site prototype designed to evaluate scalability and application server performance. RUBiS has been used by other researchers to approximate realistic Internet Services [12–14].

We use the PHP implementation of RUBiS running Apache as the web server and MySQL as the database. For consistent results, we re-create the database and pre-populate it with 100,000 users and items before each experiment. We use the default read-write transaction table for the workload that exercises all aspects of the system such as adding new items, placing bids, adding comments, viewing and browsing the database. The RUBiS workload generators warm up for 60 seconds, followed by a session run time of 600 seconds and ramp down for 60 seconds.

We emulate a topology of 40 nodes consisting of 8 database servers, 16 web servers and 16 workload generators as shown in Figure 4. A 100 Mbps network link connects two replicas of the service spread across the wide-area at two sites. Within a site, 1 Gbps links connect all components. For reliability, half of the web servers at each site use the database servers in the other site. There is one load generator per web server and all load generators share a 100 Mbps access link. Each system component (servers, workload generators) runs in its own Xen VM.

We now evaluate DieCast’s ability to predict the behavior of this RUBiS configuration using fewer resources. Figures 8(a) and 8(b) compare the baseline performance with the scaled system for overall system throughput and average response time (across all client-webserver combinations) on the y-axis as a function of number of simultaneous clients (offered load) on the x-axis. In both cases, the performance of the scaled service closely tracks that of the baseline. We also show the performance for the “No DieCast” configuration: regular VM multiplexing with no DieCast-scaling. Without DieCast to offset the resource contention, the aggregate throughput drops with a substantial increase in response times. Interestingly, for one of our initial tests, we ran with an unintended mis-configuration of the RUBiS database: the workload had commenting-related operations enabled, but the relevant tables were missing from the database. This led to an approximately 25% error rate

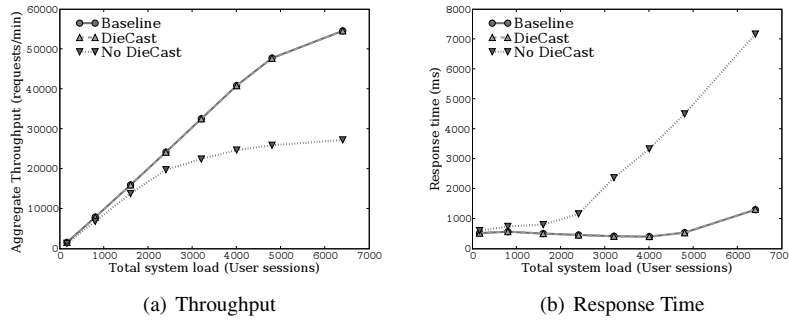


Figure 8: Comparing RUBiS application performance: Baseline vs. DieCast.

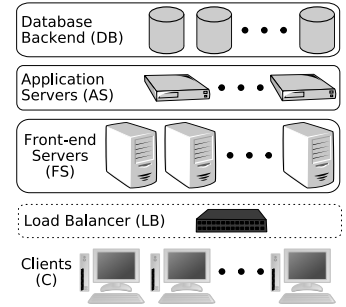
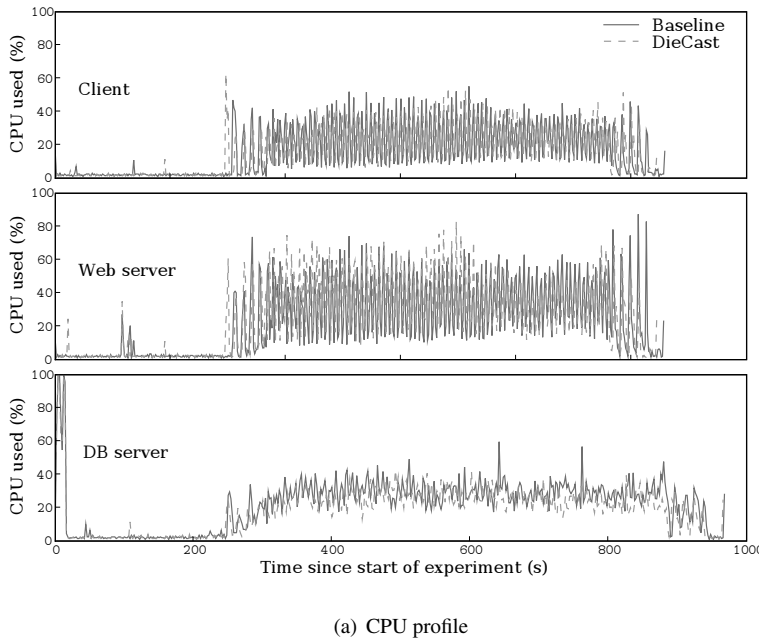
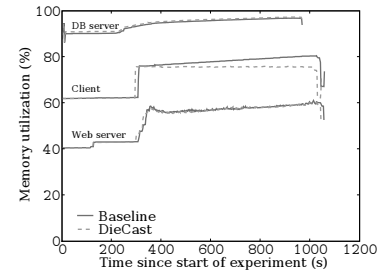


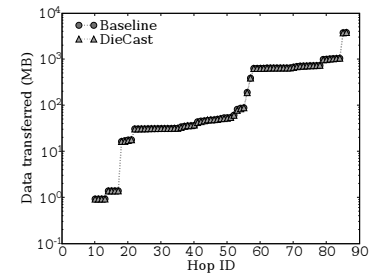
Figure 10: Architecture of Isaac.



(a) CPU profile



(b) Memory profile



(c) Network profile

Figure 9: Comparing resource utilization for RUBiS: DieCast can accurately emulate the baseline system behavior.

with similar timings in the responses to clients in both the baseline and DieCast configurations. These types of configuration errors are one example of the types of testing that we wish to enable with DieCast.

Next, Figures 9(a) and 9(b) compare CPU and memory utilizations for both the scaled and unscaled experiments as a function of time for the case of 4800 simultaneous user sessions: we pick one node of each type (DB server, Web server, load generator) at random from the baseline, and use the same three nodes for comparison with DieCast. One important question is whether the average performance results in earlier figures hide significant incongruities in per-request performance. Here, we see that resource utilization in the DieCast-scaled experiments closely tracks the utilization in the baseline on a per-node and per-tier (client, web server, database) basis. Similarly, Figure 9(c) compares the network utilization of individual links in the topology for the baseline and DieCast-scaled experiment. We sort the links by the

amount of data transferred per link in the baseline case. This graph demonstrates that DieCast closely tracks and reproduces variability in network utilization for various hops in the topology. For instance, hops 86 and 87 in the figure correspond to access links of clients and show the maximum utilization, whereas individual access links of Webservers are moderately loaded.

4.4 Exploring DieCast Accuracy

While we were encouraged by DieCast's ability to scale RUBiS and BitTorrent, they represent only a few points in the large space of possible network service configurations, for instance, in terms of the ratios of computation to network communication to disk I/O. Hence, we built Isaac, a configurable multi-tier network service to stress the DieCast methodology on a range of possible configurations. Figure 10 shows Isaac's architecture. Requests originating from a client (*C*) travel to a unique front end server (*FS*) via a load balancer (*LB*). The FS makes

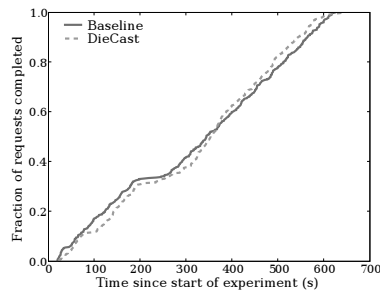


Figure 11: Request completion time.

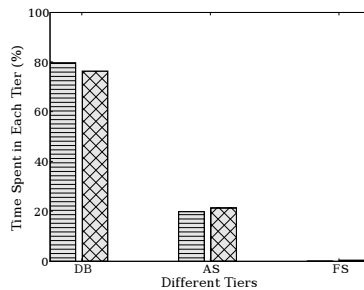


Figure 12: Tier-breakdown.

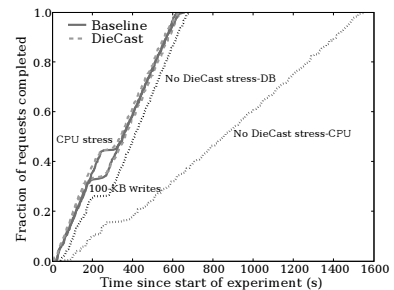


Figure 13: Stressing DB/CPU.

a number of calls to other services through application servers (*AS*). These application servers in turn may issue read and write calls to a database back end (*DB*) before building a response and transmitting it back to the front end server, which finally responds to the client.

Isaac is written in Python and allows configuring the service to a given interconnect topology, computation, communication, and I/O pattern. A configuration describes, on a per request class basis, the computation, communication, and I/O characteristics across multiple service tiers. In this manner, we can configure experiments to stress different aspects of a service and to independently push the system to capacity along multiple dimensions. We use MySQL for the database tier to reflect a realistic transactional storage tier.

For our first experiment, we configure Isaac with four DBs, four ASs, four FSs and 28 clients. The clients generate requests, wait for responses, and sleep for some time before generating new requests. Each client generates 20 requests and each such request touches five ASs (randomly selected at run time) after going through the FS. Each request from the AS involves 10 reads from and 2 writes to a database each of size 1KB. The database server is also chosen randomly at runtime. Upon completing its database queries, each AS computes 500 SHA-1 hashes of the response before sending it back to the FS. Each FS then collects responses from all five AS's and finally computes 5,000 SHA-1 hashes on the concatenated results before replying to the client. In later experiments, we vary both the amount of computation and I/O to quantify sensitivity to varying resource bottlenecks

We perform this 40-node experiment both with and without DieCast. For brevity, we do not show the results of initial tests validating DieCast accuracy (in all cases, performance matched closely in both the dilated and baseline case). Rather, we run a more complex experiment where a subset of the machines fail and then recover. Our goal is to show that DieCast can accurately match application performance before the failure occurs, during the failure scenario, and the application's recovery behavior. After 200 seconds, we fail half of the database servers (chosen at random) by stopping MySQL servers

on the corresponding nodes. As a result, client requests accessing failed databases will not complete, slowing the rate of completed requests. After one minute of downtime, we restart the MySQL server and soon after we expect to see the request completion rate to regain its original value. Figure 11 shows fraction of requests completed on the Y-axis as a function of time since the start of the experiment on the X-axis. DieCast closely matches the baseline application behavior with a dilation factor of 10. We also compare the percentage of time spent in each of the three tiers of Isaac averaged across all requests. Figure 12 shows that in addition to the end-to-end response time, DieCast closely tracks the system behavior on a per-tier basis.

Encouraged by the results of the previous experiment, we next attempt to saturate individual components of Isaac to explore the limits of DieCast's accuracy. First, we evaluate DieCast's ability to scale network services when database access dominates per-request service time. Figure 13 shows the completion time for requests, where each service issues a 100-KB (rather than 1-KB) write to the database with all other parameters remaining the same. This amounts to a total of 1 MB of database writes for every request from a client. Even for these larger data volumes, DieCast faithfully reproduces system performance. While for this workload, we are able to maintain good accuracy, the evaluation of disk dilation summarized in Figure 2(c) suggests that there will certainly be points where disk dilation inaccuracy will affect overall DieCast accuracy.

Next, we evaluate DieCast accuracy when one of the components in our architecture saturates the CPU. Specifically, we configure our front-end servers such that prior to sending each response to the client, they compute SHA-1 hashes of the response 500,000 times to artificially saturate the CPU of this tier. The results of this experiment too are shown in Figure 13. We are encouraged overall as the system does not significantly diverge even to the point of CPU saturation. For instance, the CPU utilization for nodes hosting the FS in this experiment varied from 50 – 80% for the duration of the experiment and even under such conditions DieCast closely matched

the baseline system performance. The “No DieCast” lines plot the performance of the stress-DB and stress-CPU configurations with regular VM multiplexing without DieCast-scaling. As with BitTorrent and RUBiS, we see that without DieCast, the test infrastructure fails to predict the performance of the baseline system.

5 Commercial System Evaluation

While we were encouraged by DieCast’s accuracy for the applications we considered in Section 4, all of the experiments were designed by DieCast authors and were largely academic in nature. To understand the generality of our system, we consider its applicability to a large-scale commercial system.

Panasas [4] builds scalable storage systems targeting Linux cluster computing environments. It has supplied solutions to several government agencies, oil and gas companies, media companies and several commercial HPC enterprises. A core component of Panasas’s products is the PanFS parallel filesystem (henceforth referred to as PanFS): an object-based cluster filesystem that presents a single, cache coherent unified namespace to clients.

To meet customer requirements, Panasas must ensure its systems can deliver appropriate performance under a range of client access patterns. Unfortunately, it is often impossible to create a test environment that reflects the setup at a customer site. Since Panasas has several customers with very large super-computing clusters and limited test infrastructure at its disposal, its ability to perform testing at scale is severely restricted by hardware availability; exactly the type of situation DieCast targets. For example, the Los Alamos National Lab has deployed PanFS with its Roadrunner peta-scale super computer [5]. The Roadrunner system is designed to deliver a sustained performance level of one petaflop at an estimated cost of \$90 million. Because of the tremendous scale and cost, Panasas cannot replicate this computing environment for testing purposes.

Porting Time Dilation. In evaluating our ability to apply DieCast to PanFS, we encountered one primary limitation. PanFS clients use a Linux kernel module to communicate with the PanFS server. The client-side code runs on recent versions of Xen, and hence, DieCast supported them with no modifications. However, the PanFS server runs in a custom operating system derived from an older version of FreeBSD that does not support Xen. The significant modifications to the base FreeBSD operating system made it impossible to port PanFS to a more recent version of FreeBSD that does support Xen. Ideally, it would be possible to simply encapsulate the PanFS server in a fully virtualized Xen VM. However, recall that this requires virtualization support in the processor

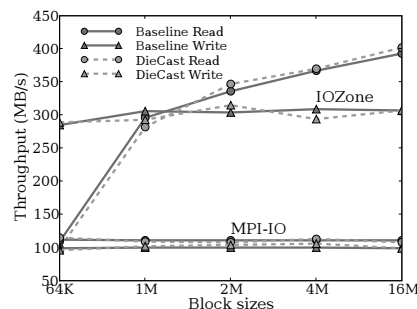


Figure 14: Validating DieCast on PanFS.

which was unavailable in the hardware Panasas was using. Even if we had the hardware, Xen did not support FreeBSD on FV VMs until recently due to a well known bug [2]. Thus, unfortunately we could not easily employ the existing time dilation techniques with PanFS on the server side. However, since we believe DieCast concepts are general and not restricted to Xen, we took this opportunity to explore whether we could modify the PanFS OS to support DieCast, without any virtualization support.

To implement time dilation in the PanFS kernel, we scale the various time sources, and consequently, the wall clock. The TDF can be specified at boot time as a kernel parameter. As before, we need to scale down resources available to PanFS such that its perceived capacity matches the baseline.

For scaling the network, we use Dummynet [27], which ships as part of the PanFS OS. However, there was no mechanism for limiting the CPU available to the OS, or to slow the disk. The PanFS OS does not support non work-conserving CPU allocation. Further, simply modifying the CPU scheduler for user processes is insufficient because it would not throttle the rate of kernel processing. For CPU dilation, we had to modify the kernel as follows. We created a CPU-bound task, (`idle`), in the kernel and we statically assigned it the highest scheduling priority. We scale the CPU by maintaining the required ratio between the run times of the `idle` task and all remaining tasks. If the `idle` task consumes sufficient CPU, it is removed from the run queue and the regular CPU scheduler kicks in. If not, the scheduler always picks the `idle` task because of its priority.

For disk dilation, we were faced by the complication that multiple hardware and software components interact in PanFS to service clients. For performance, there are several parallel data paths and many operations are either asynchronous or cached. Accurately implementing disk dilation would require accounting for all of the possible code paths as well as modeling the disk drives with high fidelity. In an ideal implementation, if the physical service time for a disk request is s and the TDF is t , then the request should be delayed by time $(t - 1)s$ such that the total physical service time becomes $t \times s$, which under

dilation would be perceived as the desired value of s .

Unfortunately, the Panasas operating system only provides coarse-grained kernel timers. Consequently, sleep calls with small durations tend to be inaccurate. Using a number of micro-benchmarks, we determined that the smallest sleep interval that could be accurately implemented in the PanFS operating system was 1 ms.

This limitation affects the way disk dilation can be implemented. For I/O intensive workloads, the rate of disk requests is high. At the same time, the service time of each request is relatively modest. In this case, delaying each request individually is not an option, since the overhead of invoking sleep dominates the injected delay and gives unexpectedly large slowdowns. Thus, we chose to aggregate delays across some number of requests whose service time sums to more than 1 ms and periodically inject delays rather than injecting a delay for each request. Another practical limitation is that it is often difficult to accurately bound the service time of a disk request. This is a result of the various I/O paths that exist: requests can be synchronous or asynchronous, they can be serviced from the cache or not and so on.

While we realize that this implementation is imperfect, it works well in practice and can be automatically tuned for each workload. A perfect implementation would have to accurately model the low level disk behavior and improve the accuracy of the kernel sleep function. Because operating systems and hardware will increasingly support native virtualization, we feel that our simple disk dilation implementation targeting individual PanFS workloads is reasonable in practice to validate our approach.

Validation We first wish to establish DieCast accuracy by running experiments on bare hardware and comparing them against DieCast-scaled virtual machines. We start by setting up a storage system consisting of an PanFS server with 20 disks of capacity 250GB each (5TB total storage). We evaluate two benchmarks from the standard bandwidth test suite used by Panasas. The first benchmark involves 10 clients (each on a separate machine) running IOZone [23]. The second benchmark uses the Message Passing Interface (MPI) across 100 clients (again, on separate machines) [28].

For DieCast scaling, we repeat the experiment with our modifications to the PanFS server configured to enforce a dilation factor of 10. Thus, we allocate 10% of the CPU to the server and dilate the network using Dummynet to 10% of the physical bandwidth and 10 times the latency (to preserve the bandwidth-delay product). On the client side, we have all clients running in separate virtual machines (10 VMs per physical machine), each receiving 10% of the CPU with a dilation factor of 10.

Figure 14 plots the aggregate client throughput for both experiments on the y-axis as a function of the data block size on the x-axis. Circles mark the read

Aggregate Throughput	Number of clients		
	10	250	1000
Write	370 MB/s	403 MB/s	398 MB/s
Read	402 MB/s	483 MB/s	424 MB/s

Table 1: Aggregate read/write throughputs from the IOZone benchmark with block size 16M. PanFS performance scales gracefully with larger client populations.

throughput while triangles mark write throughput. We use solid lines for the baseline and dashed lines for the DieCast-scaled configuration. For both reads and writes, DieCast closely follows baseline performance, never diverging by more than 5% even for unusually large block sizes.

Scaling With sufficient faith in the ability of DieCast to reproduce performance for real-world application workloads we next aim to push the scale of the experiment beyond what Panasas can easily achieve with their existing infrastructure.

We are interested in the scalability of PanFS as we increase the number of clients by two orders of magnitude. To achieve this, we design an experiment similar to the one above, but this time we fix the block size at 16MB and vary the number of clients. We use 10 VMs each on 25 physical machines to support 250 clients to run the IOZone benchmark. We further scale the experiment by using 10 VMs each on 100 physical machines to go up to 1000 clients. In each case, all VMs are running at a TDF of 10. The PanFS server also runs at a TDF of 10 and all resources (CPU, network, disk) are scaled appropriately. Table 1 shows that the performance of PanFS with increasing client population. Interestingly, we find relatively little increase in throughput as we increase the client population. Upon investigating further, we found that a single PanFS server configuration is limited to 4 Gb/s (500 MB/s) of aggregate bisection bandwidth between the servers and clients (including any IP and filesystem overhead). While our network emulation accurately reflected this bottleneck, we did not catch the bottleneck until we ran our experiments. We leave a performance evaluation when removing this bottleneck to future work.

We would like to emphasize that prior to our experiment, Panasas had been unable to perform experiments at this scale. This is in part due to the fact that such a large number of machines might not be available at any given time for a single experiment. Further, even if machines are available, blocking a large number of machines results in significant resource contention because several other smaller experiments are then blocked on availability of resources. Our experiments demonstrate that DieCast can leverage existing resources to work around

these types of problems.

6 DieCast Usage Scenarios

In this section, we discuss DieCast’s applicability and limitations for testing large-scale network services in a variety of environments.

DieCast aims to reproduce the performance of an original system configuration and is well suited for predicting the behavior of the system under a variety of workloads. Further, because the test system can be subject to a variety of realistic and projected client access patterns, DieCast may be employed to verify that the system can maintain the terms of Service Level Agreements (SLA).

It runs in a controlled and partially emulated network environment. Thus, it is relatively straightforward to consider the effects of revamping a service’s network topology (e.g., to evaluate whether an upgrade can alleviate a communication bottleneck). DieCast can also systematically subject the system to failure scenarios. For example, system architects may develop a suite of faultloads to determine how well a service maintains response times, data quality, or recovery time metrics. Similarly, because DieCast controls workload generation it is appropriate for considering a variety of attack conditions. For instance, it can be used to subject an Internet service to large-scale Denial-of-Service attacks. DieCast may enable evaluation of various DOS mitigation strategies or software architectures.

Many difficult-to-isolate bugs result from system configuration errors (e.g., at the OS, network, or application level) or inconsistencies that arise from “live upgrades” of a service. The resulting faults may only manifest as errors in a small fraction of requests and even then after a specific sequence of operations. Operator errors and mis-configurations [22,24] are also known to account for a significant fraction of service failures. DieCast makes it possible to capture the effects of mis-configurations and upgrades before a service goes live.

At the same time, DieCast will not be appropriate for certain service configurations. As discussed earlier, DieCast is unable to scale down the memory or storage capacity of a service. Services that rely on multi-petabyte data sets or saturate the physical memories of all of their machines with little to no cross-machine memory/storage redundancy may not be suitable for DieCast testing. If system behavior depends heavily on the behavior of the processor cache, and if multiplexing multiple VMs onto a single physical machine results in significant cache pollution, then DieCast may under-predict the performance of certain application configurations.

DieCast may change the fine-grained timing of individual events in the test system. Hence, DieCast may not be able to reproduce certain race conditions or timing errors in the original service. Some bugs, such as memory

leaks, will only manifest after running for a significant period of time. Given that we inflate the amount of time required to carry out a test, it may take too long to isolate these types of errors using DieCast.

Multiplexing multiple virtual machines onto a single physical machine, running with an emulated network, and dilating time will introduce some error into the projected behavior of target services. This error has been small for the network services and scenarios we evaluate in this paper. In general however, DieCast’s accuracy will be service and deployment-specific. We have not yet established an overall limit to DieCast’s scaling ability. In separate experiments not reported in this paper, we have successfully run with scaling factors of 100. However, in these cases, the limitation of time itself becomes significant. Waiting 10 times longer for an experiment to configure is often reasonable, but waiting 100 times longer becomes difficult.

Some services employ a variety of custom hardware, such as load balancing switches, firewalls, and storage appliances. In general, it may not be possible to scale such hardware in our test environment. Depending on the architecture of the hardware, one approach is to wrap the various operating systems for such cases in scaled virtual machines. Another approach is to run the hardware itself and to build custom wrappers to intercept requests and responses, scaling them appropriately. A final option is to run such hardware unscaled in the test environment, introducing some error in system performance. Our work with PanFS shows that it is feasible to scale unmodified services into the DieCast environment with relatively little work on the part of the developer.

7 Related Work

Our work builds upon previous efforts in a number of areas. We discuss each in turn below.

Testing scaled systems SHRiNK [25] is perhaps most closely related to DieCast in spirit. SHRiNK aims to evaluate the behavior of faster networks by simulating slower ones. For example, their “scaling hypothesis” states that the behavior of 100Mbps flows through a 1Gbps pipe should be similar to 10Mbps through a 100Mbps pipe. When this scaling hypothesis holds, it becomes possible to run simulations more quickly and with a lower memory footprint. Relative to this effort, we show how to scale fully operational computer systems, considering complex interactions among CPU, network, and disk spread across many nodes and topologies.

Testing through Simulation and Emulation One popular approach to testing complex network services is through building a simulation model of system behavior under a variety of access patterns. While such simulations are valuable, we argue that simulation is best suited to understanding coarse-grained performance character-

istics of certain configurations. Simulation is less suited to configuration errors or to capturing the effects of unexpected component interactions, failures, etc.

Superficially, emulation techniques (e.g. Emulab [34] or ModelNet [31]), offer a more realistic alternative to simulation because they support running unmodified applications and operating systems. Unfortunately, such emulation is limited by the capacity of the available physical hardware and hence is often best suited to considering wide-area network conditions (with smaller bisection bandwidths) or smaller system configurations. For instance, multiplexing 1000 instances of an overlay across 50 physical machines interconnected by Gigabit Ethernet may be feasible when evaluating a file sharing service on clients with cable modems. However, the same 50 machines will be incapable of emulating the network or CPU characteristics of 1000 machines in a multi-tier network service consisting of dozens of racks and high-speed switches.

Time Dilation DieCast leverages earlier work on Time Dilation [19] to assist with scaling the network configuration of a target service. This earlier work focused on evaluating network protocols on next-generation networking topologies, e.g., the behavior on TCP on 10Gbps Ethernet while running on 1Gbps Ethernet. Relative to this previous work, DieCast improves upon time dilation to scale *down* a particular network configuration. In addition, we demonstrate that it is possible to trade time for compute resources while accurately scaling CPU cycles, complex network topologies, and disk I/O. Finally, we demonstrate the efficacy of our approach end-to-end for complex, multi-tier network services.

Detecting Performance Anomalies There have been a number of recent efforts to debug performance anomalies in network services, including Pinpoint [14], MagPie [9], and Project 5 [8]. Each of these initiatives analyzes the communication and computation across multiple tiers in modern Internet services to locate performance anomalies. These efforts are complementary to ours as they attempt to locate problems in deployed systems. Conversely, the goal of our work is to test particular software configurations at scale to locate errors before they affect a live service.

Modeling Internet Services Finally, there have been many efforts to model the performance of network services to, for example, dynamically provision them in response to changing request patterns [16, 30] or to reroute requests in the face of component failures [12]. Once again, these efforts typically target already running services relative to our goal of testing service configurations. Alternatively, such modeling could be used to feed simulations of system behavior or to verify at a coarse granularity DieCast performance predictions.

8 Conclusion

Testing network services remains difficult because of their scale and complexity. While not technically or economically feasible, a comprehensive evaluation would require running a test system identically configured to and at the same scale as the original system. Such testing should enable finding performance anomalies, failure recovery problems, and configuration errors under a variety of workloads and failure conditions before triggering corresponding errors during live runs.

In this paper, we present a methodology and framework to enable system testing to more closely match both the configuration and scale of the original system. We show how to multiplex multiple virtual machines, each configured identically to a node in the original system, across individual physical machines. We then dilate individual machine resources, including CPU cycles, network communication characteristics, and disk I/O, to provide the illusion that each VM has as much computing power as corresponding physical nodes in the original system. By trading time for resources, we enable more realistic tests involving more hosts and more complex network topologies than would otherwise be possible on the underlying hardware. While our approach does add necessary storage and multiplexing overhead, an evaluation with a range of network services, including a commercial filesystem, demonstrates our accuracy and the potential to significantly increase the scale and realism of testing network services.

Acknowledgements

The authors would like to thank Tejasvi Aswathanarayana, Jeff Butler and Garth Gibson at Panasas for their guidance and support in porting DieCast to their systems. We would also like to thank Marvin McNett and Chris Edwards for their help in managing some of the infrastructure. Finally, we would like to thank our shepherd Steve Gribble, and our anonymous reviewers for their time and insightful comments—they helped tremendously in improving the paper.

References

- [1] BitTorrent. <http://www.bittorrent.com>.
- [2] FreeBSD bootloader stops with BTX halted in hvm domU. http://bugzilla.xensource.com/bugzilla/show_bug.cgi?id=622.
- [3] Netem. <http://linux-net.osdl.org/index.php/Netem>.
- [4] Panasas. <http://www.panasas.com>.
- [5] Panasas ActiveScale Storage Cluster Will Provide I/O for World's Fastest Computer. http://panasas.com/press_release_111306.html.
- [6] RUBiS. <http://rubis.objectweb.org>.

- [7] VMware appliances. <http://www.vmware.com/vmtn/appliances/>.
- [8] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.
- [9] P. Barham, A. Doelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.
- [10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.
- [11] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 2003.
- [12] J. M. Blanquer, A. Batchelli, K. Schauer, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.
- [13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2002.
- [14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, 2002.
- [15] Y.-C. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proceedings of the USENIX Annual Technical Conference*, 2004.
- [16] R. Doyle, J. Chase, O. Asad, W. Jen, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.
- [17] G. R. Ganger and contributors. The DiskSim Simulation Environment. <http://www.pdl.cmu.edu/DiskSim/index.html>.
- [18] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Network Services with an Accurate 1/10 Scale Model. Technical Report CS2007-0910, University of California, San Diego, 2007.
- [19] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, G. M. Voelker, and A. Vahdat. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.
- [20] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.
- [21] J. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *Proceedings of the first EuroSys Conference*, 2006.
- [22] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.
- [23] W. Norcott and D. Capps. IOzone Filesystem Benchmark. <http://www.iozone.org/>.
- [24] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.
- [25] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. SHRINK: A Method for Scalable Performance Prediction and Efficient Network Simulation. In *IEEE INFOCOM*, 2003.
- [26] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. In *SIGCOMM Computer Communications Review*, volume 33, 2003.
- [27] L. Rizzo. Dummynet and Forward Error Correction. In *Proceedings of the USENIX Annual Technical Conference*, 1998.
- [28] The MPI Forum. MPI: A Message Passing Interface. pages 878–883, Nov. 1993.
- [29] A. Tridgell. Emulating Netbench. <http://samba.org/ftp/tridge/dbench/>.
- [30] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.
- [31] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.
- [32] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.
- [33] A. Warfield, R. Ross, K. Fraser, C. Limpach, and H. Steven. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*.
- [34] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

D³S: Debugging Deployed Distributed Systems

Xuezheng Liu[†] Zhenyu Guo[†] Xi Wang[‡] Feibo Chen[¶]
Xiaochen Lian[§] Jian Tang[†] Ming Wu[†] M. Frans Kaashoek^{*} Zheng Zhang[†]

[†]Microsoft Research Asia [‡]Tsinghua University
[¶]Fudan University [§]Shanghai Jiaotong University ^{*}MIT CSAIL

Abstract

Testing large-scale distributed systems is a challenge, because some errors manifest themselves only after a distributed sequence of events that involves machine and network failures. D³S is a checker that allows developers to specify predicates on distributed properties of a deployed system, and that checks these predicates while the system is running. When D³S finds a problem it produces the sequence of state changes that led to the problem, allowing developers to quickly find the root cause.

Developers write predicates in a simple and sequential programming style, while D³S checks these predicates in a distributed and parallel manner to allow checking to be scalable to large systems and fault tolerant. By using binary instrumentation, D³S works transparently with legacy systems and can change predicates to be checked at runtime. An evaluation with 5 deployed systems shows that D³S can detect non-trivial correctness and performance bugs at runtime and with low performance overhead (less than 8%).

1 Introduction

Distributed systems are evolving rapidly from simple client/server applications to systems that are spread over many machines, and these systems are at the heart of today's Internet services. Because of their scale these systems are difficult to develop, test, and debug. These systems often have bugs that are difficult to track down, because the bugs exhibit themselves only after a certain sequence of events, typically involving machine or network failures, which are often difficult to reproduce.

The approach to debugging used in practice is for developers to insert print statements to expose local state, buffer the exposed state, and periodically send the buffers to a central machine. The developer then writes a script to parse the buffers, to order the state of each machine in a global snapshot, and to check for incorrect behavior.

This approach is effective both during development and deployment, but has some disadvantages for a developer: the developer must write code to record the state of each machine and order these states into a globally-consistent snapshot. The developer must anticipate what state to record; an implementation monitoring too much state may slow down the deployed system, while monitoring too little may miss detection of incorrect behavior. The developer may need to distribute the checking across several machines, because a central checker may be unable to keep up with a system deployed on many machines—an application we worked with produced 500~1000 KB/s of monitoring data per machine, which is a small fraction (1~2%) of the total data handled by the application, but enough monitoring data as a whole that a single machine could not keep up. Finally, the developer should have a plan to approximate a globally-consistent snapshot when some processes that are being checked fail, and should make the checking itself fault tolerant.

Although many tools have been proposed for simplifying debugging of distributed or parallel applications (see Section 7), we are unaware of a tool that removes these disadvantages. To fill that need, we propose D³S, a tool for debugging deployed distributed systems, which automates many aspects of the manual approach, allows runtime checking to scale to large systems, and makes the checking fault tolerant.

Using D³S, a developer writes functions that check distributed predicates. A predicate is often a distributed invariant that must hold for a component of the system or the system as a whole (e.g., “no two machines should hold the same lock exclusively”). D³S compiles the predicates and dynamically injects the compiled libraries to the running processes of the system and additional verifier processes that check the system. After injection, the processes of the system expose their states as tuples (e.g., the locks a process holds), and stream the tuples to the verifier processes for checking. When the

checking identifies a problem (e.g., two processes that hold the same lock), D³S reports the problem and the sequence of state changes that led to the problem. By using binary instrumentation, D³S can transparently monitor a deployed, legacy system, and developers can change predicates at runtime.

A key challenge in the design of D³S is to allow the developer to express easily what properties to check, yet allow the checking to use several machines so that the developer can check large systems at runtime. A second challenge is that D³S should handle failures of checking machines. A third challenge is that D³S should handle failures of processes being checked—the checkers should continue running without unnecessary false negatives or positives in the checking results. Using the lock example, suppose a client acquires a lock with a lease for a certain period but then fails before releasing the lock, and after the lease expires another client acquires the lock. The predicate that checks for double acquires should not flag this case as an error. To avoid this problem, D³S must handle machine failures when computing snapshots.

D³S's design addresses these challenges as follows. For the first challenge, D³S allows developers to organize checkers in a directed-acyclic graph, inspired by Dryad [21]. For each vertex, which represents a computation stage, developers can write a sequential C++ function for checkers of this stage; the function can reuse type declarations from the program being checked. The state tuples output by the checkers flow to the downstream vertices in the graph. During the checking, a vertex can be mapped to several verifier processes that run the checkers in parallel; in this way D³S can use multiple machines to scale runtime checking to large systems. Within this framework, D³S also incorporates sampling of the state being checked, and incremental checking. These features can make the checking more lightweight.

For the second challenge, D³S monitors verifier processes. When one fails, D³S starts a new verifier process and feeds it the input of the failed process. Because checkers are deterministic, D³S can re-execute the checkers with the same input.

For the third challenge, the verifier processes remove failed processes from globally-consistent snapshots before checking the snapshots. D³S uses a logical clock [24] to order the exposed state tuples, and has a well-defined notion of which processes are in the snapshot at each timestamp. For the previous lock example, D³S will not report a false positive, because the lock state acquired by the first client is removed from the snapshot at the time when the second client acquires the lock.

We have implemented D³S on Windows and used it to check several distributed systems. We were able to quickly find several intricate bugs in a semi-structured

storage system [26], a Paxos [25] implementation, a Web search engine [37], a Chord implementation [35, 1], and a BitTorrent client [2]. We also found that the burden of writing predicate checkers for these systems was small (the largest predicate we used has 210 lines of code; others are around 100 lines), and that the overhead of runtime checking was small compared to the system being checked (the largest overhead is 8% but for most cases it is less than 1%).

The main contributions of this paper are: the model for writing predicate checkers; the runtime that allows real-time checking to scale to large systems, that constructs global snapshots to avoid unnecessary false negatives and positives, and that handles failures of checking machines; and, the evaluation with 5 distributed systems.

The rest of this paper is organized as follows. Section 2 details D³S's design. Section 3 explains how D³S computes global snapshots. Section 4 describes our implementation of D³S on Windows. Section 5 presents an evaluation with several distributed systems. Sections 6 reports on the performance of D³S. Section 7 relates D³S to previous work. Section 8 summarizes our conclusions.

2 Design

We describe the design of D³S (see Figure 1). D³S compiles a predicate into a state-exposer library and a checking library. Using binary instrumentation, D³S dynamically injects the state exposer into the running processes of the system. The state exposers produce tuples describing the current state of interest, and partitions the stream of tuples among the verifier processes. The verifying processes either run on dedicated machines or on the same machines that the system runs on. The verifying processes order tuples globally, and evaluate the predicate on snapshots of tuples. If a predicate fails, D³S reports to the developer the problem and the sequence of state changes that led to the problem.

The rest of this section describes the design of D³S in more detail: how developers write predicates, how D³S inserts them into a deployed system, how D³S allows for parallel checking and stream processing for checking predicates in a scalable and efficient manner.

2.1 Writing predicates

To illustrate the ease with which a developer can write a predicate, we will describe a predicate written in C++ that we have used to check the distributed lock service in Boxwood [29]. This distributed lock service allows clients to acquire multiple-reader single-writer locks. A lock can be held in either `Shared` or `Exclusive` mode. A critical property of the service is that the lock holders must be consistent, i.e., either there is one

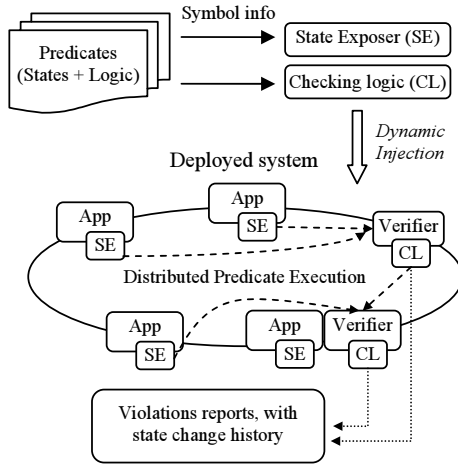


Figure 1: Overview of D³S.

Exclusive holder and no Shared holders, or there is no Exclusive holders. Because clients cache locks locally (to reduce traffic between the clients and the lock server), only the clients know the current state of a lock.

Figure 2 shows the code that the developer writes to monitor and check the properties of Boxwood’s distributed lock service. The developer organizes the predicate checking in several stages and expresses how the stages are connected in an acyclic graph; the developer describes this graph with the script part of the code. In the example there are only two stages that form a single edge with two vertices (V_0 and V_1). (Later examples in this paper have more stages.)

The vertex V_0 represents the system and the state it generates. The developer describes the state after a change as a set of tuples; in the example, each tuple has three fields of types: *client:ClientID*, *lock:LockID* and *mode:LockMode*. These types come from the header file of the lock service code, and the developer can reuse them in the script and C++ code. The tuples together express the locks and their state that a lock client is holding.

The vertex V_1 represents the computation of the lock predicate. As the script shows, V_1 takes as input the output of V_0 and generates a set of tuples, each of which has one field *conflict:LockID*. This vertex is marked as **final** to indicate it is the final stage of the checker.

The developer specifies the computation to check the predicate at vertex V_1 by writing C++ code, again reusing the types of the system being checked. In the example, the computation is the class *LockVerifier*, which is derived from the *Vertex* class and which the developer ties to V_1 using a template argument. The developer must write a method *Execute*. The D³S runtime invokes this method each time it constructs a global snapshot of tuples of the type that V_0 produces for a timestamp t ; how the runtime produces sequences of global snapshots is

```
# scripts 1. Describe computation graph with output type in each stage
V0: exposer → { (client: ClientID, lock: LockID, mode: LockMode) }
V1: V0 → { (conflict: LockID) } as final
# 2. Correlate state changes with monitored functions in app's code
after (ClientNode::OnLockAcquired) addtuple ($0->m_NodeID, $1, $2)
after (ClientNode::OnLockReleased) deltuple ($0->m_NodeID, $1, $2)

// C++ code for predicate in V1.
class LockVerifier : public Vertex< V1 > {
virtual void Execute(const V0::Collection & snapshot) {
    std::map< LockID, int > exclusive, shared; // count the lock holders
    while ( ! snapshot.eof() ) {
        // V0::Tuple is V0's output type, i.e., (ClientID, LockID, LockMode)
        V0::Tuple t = snapshot.get_next();
        if ( t.mode == EXCLUSIVE )
            exclusive[t.lock]++;
        else shared[t.lock]++;
    }
    // check conflicts and add to "output" member of Vertex.
    for (Iterator it = exclusive.begin(); it != exclusive.end(); ++ it)
        if ( it->value > 1 || (it->value == 1 && exist(shared, it->key)) )
            output.add( V1::Tuple(it->key) );
}
static Key Mapping(const V0::Tuple & t) { // map states to key space
    return t.lock;
}
};
```

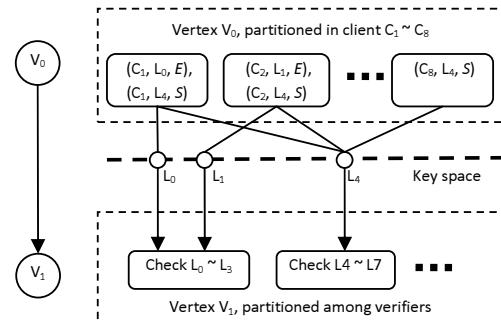


Figure 2: (a) Checking code. (b) graph and checker execution.

the topic of Section 3. In the example, *Execute* enumerates all tuples in the snapshot and tracks the number of clients holding an Exclusive and Shared lock for each lock ID. It outputs the IDs of locks that are in conflict at timestamp t .

As shown, the developer can check distributed properties by writing just sequential code that processes states in a central manner and reuses types from the system being checked. How the runtime transmits the state of multiple clients, collects the state of the clients into a globally-consistent snapshot, and checks them in parallel is hidden from the developer. This design achieves D³S’s design goals of expressiveness and simplicity.

2.2 Inserting predicates

To change what properties of a system to check, a developer can insert predicates when the system is running. The developer uses D³S’s compiler to generate

C++ code from the predicates for a state exposers and a checking logic module. The output of the compiler is two dynamically-linked libraries, one for each module, that can be attached to a running process.

D³S then disseminates and attaches the generated state exposers to every process of the system. When loaded, the state exposers rewrite the binary modules of the process, so as to add new functions that will execute either before or after the functions to be monitored (Section 4 explains the details). These new functions construct tuples in V_0 's output from memory states. For the script in Figure 2, the state exposers add two functions after `ClientNode::OnLockAcquired` and `ClientNode::OnLockReleased`, respectively, to obtain the acquired and released lock states. These functions construct tuples of the form $(\$0 \rightarrow m_NodeID, \$1, \$2)$, in which $\$i$ is the i^{th} parameter to the original function (for member functions in C++, $\$0$ is the "this" pointer). The state exposers will add or delete the constructed tuples in V_0 's output, instructed by the **addtuple** and **deltuple** keywords. The developer is allowed to embed C++ code in the script to construct tuples for V_0 in case the script needs more than the function parameters. However, we find that in most cases (all systems we checked), exposing function parameters is sufficient to monitor state changes.

The state exposers can start outputting tuples immediately after it adds all monitoring functions. Alternatively, it can start on a certain time instructed by the developer. Due to network delay, different instances of state exposers may not start at exactly the same time. This causes no problems, because the D³S verifiers will wait with running the checkers until they can construct a global snapshot.

The checking library contains the programs for vertices other than V_0 . D³S attaches them to all verifiers, and the verifiers start to process incoming tuples, run the checkers when a global snapshot is ready, and stream outputs to their next stages.

When a developer inserts a new predicate checker while the system is running, the checker may miss violations that are related to previous unmonitored history. For instance, in the lock example, if the verifier starts after a client has acquired a lock, the verifier does not know that the client already has the lock and a related violation may go undetected.

2.3 Dataflow graphs

More complex predicates than the lock example will have more complex dataflow graphs of verifiers. D³S runs vertices to process timestamp t when the input data from upstreaming vertices are ready to construct a consistent snapshot for t . After the snapshot is processed, the

output data is also labeled with t and transmitted to all downstream vertices. When all vertices has executed for t , the predicate is evaluated for t , and D³S produces the checking result from the output of the *final vertex*. Vertices can work on different timestamps simultaneously in a pipeline fashion, transparently exploiting the parallelism in the predicate.

Predicates are deterministically calculated from the exposed states. When failures happen in intermediate vertices, after recovery D³S can re-execute the same timestamp from the original exposed states in V_0 (V_0 can buffer exposed states of the timestamp, until the final stage finishes). This scheme allows D³S to handle verifier failures easily.

2.4 Partitioned execution

The D³S runtime can partition the predicate computation across multiple machines, as in Figure 2(b), with minimal guidance from the developer. Using the lock service example, to guarantee the correctness of predicate checking when the runtime partitions the computation, the tuples describing the same lock should be checked together. Similar to the Map phase in MapReduce [12], the developer expresses this requirement through the *Mapping* method, which maps output tuples to a virtual key space. The runtime then partitions the key space dynamically over several machines and runs the computation for different key ranges in parallel. Tuples mapped to a key are checked by the verifier that takes that key as input. In the example, the first and the second machine will run *Execute* for lock $0 \sim 5$ and $6 \sim 10$, respectively. Each vertex can have an independent mapping function. D³S uses a default hash function when there is no mapping function in a vertex.

A notification mechanism (details in Section 4) tells verifiers the current key assignments of their downstream vertices so that verifiers can transmit outputs to the verifiers that depend on them. If a verifier fails, its responsible input range will be taken over by other remaining verifiers. By changing the assignment of key spaces to verifiers on demand, D³S is free to add and remove verifiers, or re-balance the jobs on verifiers. This design achieves D³S's design goals of scalability and failure tolerance.

2.5 Stream processing and sampling

Often, there are only minor variations in state between consecutive timestamps. In such cases it is inefficient to transmit and process the entire snapshots at every timestamp. For this reason D³S supports stream processing, in which vertices only transmit the difference in their output compared to the last timestamp, and check the state

```

V0: exposer → { (pred: chordID, self: chordID, succ: chordID) }
V1: V0      → { (sum_range_size: int) }
V2: V1      → { (range_coverage: float) } as final
before (stabilize) deltuple ($0->leftID, $0->node.id, $0->rightID)
after (stabilize) addtuple ($0->leftID, $0->node.id, $0->rightID)

class RangeSum : public Vertex< V1 > {
    virtual void Execute(const V0::Collection & snapshot) {
        // calculate size of key range
        int sum_range_size = 0;
        while ( ! snapshot.eof() ) {
            V0::Tuple t = snapshot.get_next();
            sum_range_size += circle_distance(t.pred, t.self);
        }
        output.add( V1::Tuple( sum_range_size ) );
    }
    // incremental evaluation on delta data
    virtual void ExecuteChange(const V0::Delta & delta) {
        // calculate delta of key range size
        int delta_range_size = 0;
        while ( ! delta.eof() ) {
            V0::Tuple t; DeltaFlag flag;
            delta.get_next( t, flag );
            int sign = ( flag == DELTA_FLAG_DELETED ) ? -1 : 1;
            delta_range_size += sign * circle_distance(t.pred, t.self);
        }
        output.add( V1::Tuple( last_output + delta_range_size ) );
    }
};

class Aggregation : public Vertex< V2 > {
    virtual void Execute(const V1::Collection & snapshot) {
        // aggregate range sizes from previous vertex
        int sum = 0;
        while ( ! snapshot.eof() ) {
            sum += snapshot.get_next().sum_range_size;
        }
        output.add( V2::Tuple( sum / CIRCLE_SIZE ) );
    }
    static Key Mapping(const V1::Tuple & t) {
        return 0; // make sure all state are transmitted to single verifier
    }
};

```

Figure 3: The predicate that checks the key range coverage among Chord nodes.

incrementally. There is an optional *ExecuteChange* function to specify the logic for incremental processing.

To illustrate the use of *ExecuteChange* and dataflow graph with more vertices, we will use Chord DHT in *i3* service [35] as another example (Figure 3). Section 5.4 presents the checking results of this example.

We check the consistency of its key ranges among Chord nodes. Every Chord node exposes the ring information as tuples with three fields: *pred*, *self* and *succ*, which indicate the *chordID* of the node’s predecessor, itself and the successor, respectively. According to the *i3*-Chord implementation, the key range assigned to the node is [*pred*, *self*). The key range predicate computes the aggregate key range held by current nodes, relative to the entire ID space. In ideal case where the Chord ring is correct, this value should be 100%. Below 100% in-

dicates “holes” while above 100% indicates overlaps in key ranges.

For the key range predicate in Chord, we use three vertices $V_0 \rightarrow V_1 \rightarrow V_2$. V_0 represents state exposers that outputs states from Chord nodes. Every state represents the neighborhood of a Chord node in the ring. The second vertex V_1 calculates the sum of range sizes in *Execute* from received states from all Chord nodes.

ExecuteChange shows incremental execution. It receives only the difference of two consecutive snapshots, and uses its last output as the base of execution. This avoids most of the redundant transmission and processing on unchanged states, reducing the overhead in both state exposers and verifiers.

To make the Chord checker scalable, we partition the execution of V_1 to multiple verifiers, each verifier taking only a subset of the states. Therefore, we need a third vertex V_2 to aggregate the outputs from all verifiers in V_1 , i.e., the sum of key ranges in each partition. It calculates the relative range coverage as final output of the predicate. We use one verifier in the final vertex, and the verifier communicates with verifiers in V_1 . This algorithm is essentially a 2-level aggregation tree; more levels will further improve scalability and pipelining.

Beside stream processing, developers can use sampling to further reduce overhead. Developers can check only sampled states in each vertex. To achieve this, D³S allows verifiers to take as input only a portion of the key space for some vertices. These vertices process only the states that are mapped to covered keys. Tuples mapped to uncovered key space are dropped at the producer side. In addition, developers can check only sampled timestamps. This can be done because D³S can stop checking in the middle of system execution and restart predicate checking at later global snapshots.

With sampling, D³S can use a few verifiers to check a large-scale system in probabilistic manner. For instance, to check the consistency of Chord key range, D³S can randomly sample a number of keys at different time and check that each sampled key has exact one holder (see Section 5.4), instead of checking the entire key space all the time. This approach makes the checking more lightweight, at the risk of having false negatives (i.e., missed violations).

2.6 Discussion

The two examples check predicates for safety properties only. For liveness properties, which should *eventually* be true, a violation often implies only that the system is in fluctuating status, rather than a bug. Similar to our previous work [27], a D³S user can specify a timeout threshold plus stability measures in the predicate to filter out false alarms for liveness violations.

In theory, D³S is capable of checking any property specified on a finite length of consecutive snapshots. To be used in practice, D³S should impose negligible overhead on the system being checked, and be capable of checking large-scale systems. Based on our experience in Section 6, the overhead of the system is small in most cases, because we need to expose only states that are relevant to the predicate, and can omit other states. As a result, a state exposor consumes a tiny fraction of CPU and I/O compared with actual payloads of the system.

The scalability of checking depends on the predicate logic. When the predicate can be partitioned (as shown in our examples), a developer can add more verifiers to check larger systems. For such cases, the dataflow programming model effectively exploits parallelism within and among stages. However, some properties cannot be partitioned easily (e.g., deadlock detection that looks for cycles in the lock dependency graph). In such cases, the developer must write more sophisticated predicates to improve the scalability. For instance, the developer can add an early stage to filter locks that changed state recently, and have the final stage check only the locks that have been in the acquired state for a long time. By this means the final verifier avoids checking most correct locks, while still catching all deadlocks.

3 Global Snapshots

This section explains how D³S constructs global snapshots and how accurate the predicates are under failures. Because of machine failures, snapshots might be incomplete, missing the state of failed machines, which may lead to false positives and false negatives when the system is reconfiguring.

3.1 Snapshots

We model the execution of the system being checked as a sequential state machine that traverses a sequence of consistent snapshots with global timestamps. Assume we have an increasing timestamp sequence $\mathcal{T} = \{t_0, t_1, \dots\}$, where $t_i \in \mathcal{T}$ is a timestamp for $i \in \mathbb{N}$. The *membership* at timestamp t is the set of live processes at t , denoted by $M(t)$. For a process $p \in M(t)$, we use $S_p(t)$ to denote its local state at timestamp t . A *consistent snapshot* at t , denoted by $\pi(t)$, is the collection of states from all live processes at t , i.e., $\pi(t) = \bigcup_{p \in M(t)} S_p(t)$. With this notation, the system goes through a sequence of consistent snapshots, denoted by $\Pi = \{\pi(t_i), i = 0, 1, \dots\}$. D³S checks properties defined over these global snapshots.

To construct a global snapshot we need a global timestamp, which D³S provides in a standard manner using a logical clock [24]. Each process maintains a logical clock, which is an integer initialized to 0. Each time

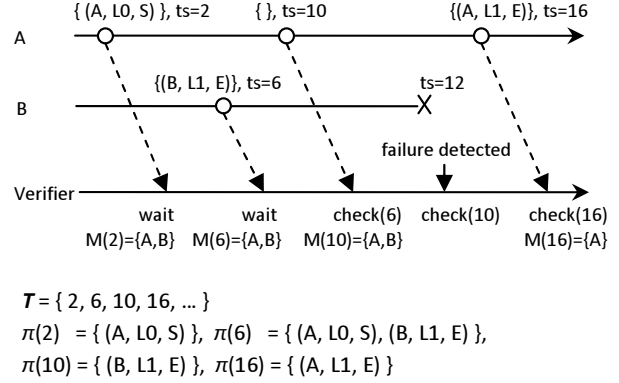


Figure 4: Predicate checking for consistency of distributed locks. Two processes A and B expose their states in the form of $\{(ClientID, LockID, Mode)\}$ (E for Exclusive and S for Shared). \mathcal{T} is the sequence of timestamps and $\pi(t)$ is the snapshot for timestamp t . Given a failure detector that outputs membership for every timestamp, the verifier can decide whether a complete snapshot is obtained for checking.

a process reads its logical clock (e.g., to timestamp its state on a state change), it increases the logical clock by 1. Each time the process sends a message, it attaches its logical clock to the message. On receiving a message, the receiving processes sets its logical clock to the maximum of its local logical clock and the clock attached to the message. This way the D³S runtime preserves happens-before relationships, can order all tuples in a consistent total order, and can construct snapshots.

Figure 4 illustrates the memberships and snapshots of the lock checking example. Process A and B are lock clients being checked, and they expose their state changes. Every state change produces a set of $(ClientID, LockID, Mode)$ tuples that represent all current locks the process holds. The state changes happen at disjoint logical times $\{2, 10, 16\}$ and $\{6\}$, respectively. In addition, process B crashes at logical time 12.

If process p exposes two set of tuples at timestamp t_1 and t_2 , for any timestamp t between t_1 and t_2 , $S_p(t) = S_p(t_1)$. For example, $S_A(6) = S_A(2) = \{(A, L0, Shared)\}$. Therefore, given $M(6) = \{A, B\}$, the snapshot $\pi(6) = S_A(6) \cup S_B(6) = S_A(2) \cup S_B(6)$.

3.2 Predicates

We model a predicate as a function defined over a finite number of consecutive snapshots. The number of consecutive snapshots needed is called the *window size* of a predicate. Specifically, a predicate P with window size n is a function evaluated for every timestamp in \mathcal{T} , $P(t_i) = F(\pi(t_{i-n+1}), \pi(t_{i-n+2}), \dots, \pi(t_i))$ for some $n \geq 1$, where F is a user-specified function. With this definition, a predicate can depend only on a recent

time window of snapshots, and thus can be checked in the middle of system running. In our experience, all useful properties can be checked with only a recent time window of snapshots.

In the lock example, the checked property is that at any time t_i , there is no conflict between read and write locks. This property is checked by a predicate over the current snapshot, i.e., $LockConsistency(t_i) = F(\pi(t_i))$ in which F checks whether $\forall l \in LockID$, the set $\{(c, l', m) \in \pi(t_i) | l' = l, m = Exclusive\}$ contains at most one element (Figure 2 (a) implements this function). So $LockConsistency$ is a predicate with window size 1. Predicates with multiple consecutive snapshots are useful when specifying historical properties.

3.3 Correctness of predicate checking

To verify a predicate correctly, D³S needs a *complete* snapshot, which contains the state of all processes that constitute the system at a timestamp. To construct a *complete* snapshot $\pi(t)$, D³S must know the membership $M(t)$, and the local states $S_p(t)$ for all p in $M(t)$, but $M(t)$ can change due to failures.

D³S relies on a *failure detector* to establish $M(t)$. We model the failure detector with a query interface, similar to most failure detector specifications [8]. A verifier can query for any timestamp t in \mathcal{T} , and the failure detector will return a *guess* on $M(t)$, denoted by $M'(t)$, which can be incorrect.

The verifier uses the failure detector as follows. It queries the failure detector and receives $M'(t)$. Then, the verifier waits until local states $S_p(t)$ for all $p \in M'(t)$ have been received. Then, it constructs snapshot $\pi(t)$ as $\bigcup_{p \in M'(t)} S_p(t)$. The verifier knows it has received $S_p(t)$ either when it receives it directly or when it receives two consecutive states $S_p(t_1)$ and $S_p(t_2)$ ($t_1 < t < t_2$). In the latter case the verifier infers that $S_p(t) = S_p(t_1)$.

If we would use this procedure unmodified, then D³S has a problem when p does not expose any state for a long time (i.e. $t_2 \gg t_1$). In that case, D³S is unable to construct $\pi(t)$ for any t between t_1 and t_2 , because it doesn't know if $S_p(t_1)$ is the latest state from p . There are several ways to deal with this problem; we describe the solution we have implemented. The state exposer injected to p sends periodically p 's current timestamp to the verifier. D³S uses this heartbeat as both failure detector and the notification of p 's progress. Thus the verifier receives a train of timestamps of heartbeats intermixed with the exposed state from p . When computing $\pi(t)$, D³S uses the latest received $S_p(t_1)$ as long as the largest timestamp received from p exceeds t . If the failure detector declares that p has crashed at t_2 through a heartbeat timeout, for all t between t_1 and t_2 , $\pi(t)$ uses $S_p(t_1)$. From t larger than t_2 , D³S excludes all p 's state.

Figure 4 provides an example. B exposes its latest state at 6 and then crashes at 12. Thus, $\pi(10)$ is $S_A(10) \cup S_B(6)$ (after waiting for more than the timeout threshold for new state update from B). $\pi(16)$, however, will exclude B , since D³S will decide that B has departed from the system.

3.4 Practical implications

D³S guarantees that as long as $M'(t) = M(t)$. In other words, if the failure detector outputs correctly for timestamp t , the corresponding snapshot will be complete. If a snapshot is incomplete, then a checker can produce false positives and false negatives. In practice, there is a tradeoff between quick error alerts and accuracy.

To handle process failures that can lead to incomplete snapshots, D³S must wait before constructing a snapshot. A larger waiting time T_{buf} results in larger buffer size to buffer state and delays violation detection. A too small T_{buf} , however, can result in imprecise results due to incorrect membership information. T_{buf} thus yields a knob to control the tradeoff between performance and accuracy.

The appropriate value of T_{buf} depends the failure detector D³S uses, and should be larger than the failure detector's timeout T_{out} . We derive T_{out} as follows. For machine-room systems, there is usually a separate membership service that monitor the machine status using lease mechanisms [18]. We can use the membership service (or our own heartbeats) in failure detector. T_{out} is set as the grace period of the lease (resp. heartbeat interval) plus message delay. For wide-area applications such as DHT, T_{out} is determined by the specification of the application that declares the status of a node from its neighborhood. For predicates that does not rely on strict event orders, e.g. some runtime statistics, T_{buf} can be any values that is practically reasonable.

As an example, consider cluster storage system that we have checked (see details in Section 5.1). The system is designed for in machine-room environment, and the largest observed message delay between state exposer and verifiers is less than 350ms. The keep-alive message in the system runs every 1000ms. Thus, T_{buf} should be at least 1000ms + 350ms, and we use 2000ms. This guarantees that the verifiers always check on consistent snapshots.

4 Implementation

We have implemented D³S on Windows. When compiling a predicate script, D³S extracts the types of the tuples and the actions of monitoring functions. It then generates the corresponding C++ source code, which contains the definitions of the tuple types, vertex classes, monitoring

functions, and the checking code in the predicates. D³S compiles the source code into a state exposers DLL and a checking DLL.

The state exposers and the logical clock in D³S uses WiDS BOX [19] to instrument processes being monitored. BOX is a toolkit for binary instrumentation [20]; it identifies the function addresses through symbol information, and rewrites the entries of functions in code module to redirect function calls. When injecting a DLL into a process, BOX loads the DLL into the process's address space, and redirects function calls that are interposed on to callbacks in the DLL. Because BOX provides rewrites code atomically, it does not need to suspend the threads in the process during the instrumentation.

Through the callbacks for application-level functions, the state exposers copies the exposed states into an internal buffer. States that are used by predicates are emitted to the verifiers, whereas all others are omitted. D³S buffers the states to batch data transmission. It waits until the collected states exceed 500 bytes, or 500 ms has elapsed after the last transmission.

The D³S runtime also interposes on the socket APIs for sending and receiving messages. In those callbacks, D³S updates the logical clock. D³S also adds 8 additional bytes to each message for the logical clock and some bookkeeping information, same as [27, 15].

To make the logical clock relate to real time, D³S divides a second in 1,000 logical ticks. Every second D³S checks the value of the logical clock, and if hasn't been updated a 1,000 times, D³S bumps the clock to 1,000. This gives a convenient way to specify timeout of the monitored processes (i.e., T_{out} in Section 3.4).

D³S uses reliable network transmission between state exposers and verifiers, and also among verifiers when computation graph has multiple levels.

D³S uses a central master machine to manage a partitioned key space. Each verifier periodically reports its recently verified timestamp to the master. A verifier is considered to be failed when it doesn't report within a timeout period. In such case the master re-arranges the partition of key space to make sure that every key is appropriately covered. The new partition is then broadcast to all related state exposers and verifiers. By this means the appropriate states will arrive at the new destinations.

5 Experience with Using D³S

To evaluate the effectiveness of D³S, we apply it to five complete, deployed systems, including systems that are based on production-quality code. Table 1 summarizes these checked systems, line of code in predicates (LoP), and the results, which are obtained within one month. For each of these systems, we will give sufficient descriptions of their logic and properties, and then report

the effectiveness of D³S with our debugging experience, in terms of both the benefits and lessons we gained.

Unless otherwise specified, experiments are performed on machines with dual 2 GHz Intel Xeon CPU and 4 GB memory, connected with 1 Gb Ethernet and run Windows Server 2003. For some systems we injected failures to improve testing coverage.

5.1 PacificA

PacificA [26] is a semi-structured distributed storage system. It shares with BigTable [9] the property that large tables are segmented into small pieces, which are replicated and distributed across a cluster of commodity servers. The goal of this project is to explore different design choices (e.g., semantics-agnostic vs. semantics-aware in replication), and develop a semi-structured storage system with better combination of these trade-offs.

PacificA has been developed for more than one year, and is fairly complete with around 70,000 lines of code. It is a good example of a complex system that builds on top of well-specified components, including: two-phase commit for consistent replica updates, perfect failure detector on storage nodes, replica group reconfiguration to handle node failures, and replica reconciliation to rejoin a replica. When combining these components, the whole system can have complex message sequences with bugs that are hard to detect and analyze. However, each component must be working correctly with predictable behaviors and invariants. Therefore, we use D³S to check individual components against their specifications.

Based on specifications, there are several invariants for every major component. These invariants include consistency among replicas, data integrity after recovery, and consistent membership view across a replica group. Violating an invariant may eventually leads to data loss or replica inconsistency. We specify these invariants as predicates in D³S, and check a daily used deployed instance of PacificA. The deployment stores a social graph data for a social network computing platform. It uses 8 servers as storage nodes, and another server as front-end for clients. Clients running in 4 other machines frequently query and update edge data stored in PacificA tables to complete graph computation tasks, which usually last from a day to a week. The size of original social graph data is 38 GB, while during the execution the intermediate tables exceed 1 TB in size. A PacificA machine can have 40 MB/s throughput at the peak time. We use another 3 machines to run verifiers, and have detected 3 bugs. We will explain in detail one bug in replica group reconfiguration. The bug violates the *primary invariant* (see page 3 of [26]) during failures.

In PacificA, the basic unit of data is a slice (100 MB data chunk). A slice is replicated in three storage nodes

Application	LoC	LoP	Predicates	Results
PacificA (Structured storage)	67,263	118	membership group consistency; replica consistency	3 correctness bugs
MPS (Paxos implementation)	6,993	50	consistency in consensus outputs; leader election	2 correctness bugs
Web search engine	26,036	81	unbalanced response time of indexing servers	1 performance problem
i3-Chord (DHT)	7,640	72	aggregate key range coverage; conflicting key holders	availability and consistency
libtorrent (BitTorrent client)	36,117	210	neighbor set; downloaded pieces; peer contribution rank	2 performance bugs; free riders

Table 1: Benchmarks and results

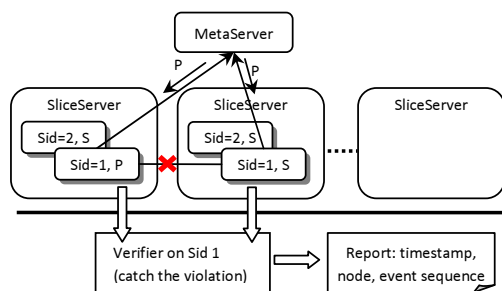


Figure 5: PacificA architecture and the bug we found.

(SliceServers), one replica being the primary and the other two being secondaries. A simplified architecture is shown in Figure 5. The primary (labeled with P) answers queries and forwards updates to the secondaries (labeled with S). These replicas monitor each other with heartbeats. When noticing a failure, the remaining replicas will issue requests to a MetaServer (master of SliceServers) for reconfiguration, and may ask to promote themselves as the primary if they think the primary is dead. The primary invariant states that at any given timestamp, there cannot be more than one primary for a slice. This is because multiple primaries can cause potential replica inconsistency during updates.

We expose replica states with tuples of the form (Sid, MachineID, {P / S}) (Figure 5), and check the number of P's for every Sid (slice identifier). Because replicas of the same slice should be checked together, we map tuples to their Sid.

As expected, the predicate found no violations in normal cases, since the system reconfigures only after a failure. To expand test coverage, we randomly injected failures to SliceServers and MetaServer and then recovered them. After dozens of tries, D³S detected a violation in a slice group, which had two primary replicas. By studying the sequence of states and the events that led to the violation, we determined that, before MetaServer crashed, it accepted a request and promote the replica to be the primary. After crash and recovery, the MetaServer forgot the previous response and accepted the second replica's request, which resulted in the second primary. This violation should have been avoided by MetaServer's failure tolerance mechanism, which logs accepted requests

to disk, and restores them at next start. However, the code uses a background thread to do batch logging, so the on-disk log may not have the last accepted request. To do it correctly, the MetaServer must flush the log *before* it sends a response.

D³S helped in detecting the bug in the following ways. First, the bug violates a distributed property that cannot be checked locally. D³S provides globally-consistent snapshots that make checking distributed properties easy. Second, contrast to postmortem log verification, D³S enforces always-on component-level specification checking, and catches rare-case bugs with enough information to determine the root cause. Without this predicate, we may still have noticed the bug when conflicting writes from two primaries would have corrupted the data. However, from this corruption it would have been difficult to determine the root cause.

We also applied D³S to an old PacificA version which has two data races. It took the developers several days to resolve them. With D³S, these bugs were caught in several hours of normal use, and state sequences provided much better hints of the root cause.

5.2 The MPS Paxos Implementation

Paxos [25] is a widely used consensus algorithm for building fault-tolerant services [29, 6]. Despite the existing literature on Paxos, implementing a complete Paxos-based service is non-trivial [7]. We checked MPS¹, a Paxos-based replicated state machine service used in production. MPS provides many features over the base Paxos protocol, e.g., it uses leader election for liveness, and perform state transfers to allow a new node to catch up with others. These additional features are necessary for ensuring progress of the service, because Paxos itself guarantees only safety but not liveness of agreement.

MPS provides an API that application code can invoke to execute arbitrary commands, with the guarantee that all nodes execute the same sequence of commands. We deploy MPS on 5 nodes, the typical configuration in cluster environment, and make each node invoke the API independently. For checking consensus protocols, testing

¹MPS is an anonymous name of a Paxos-based service system developed by Microsoft product team.

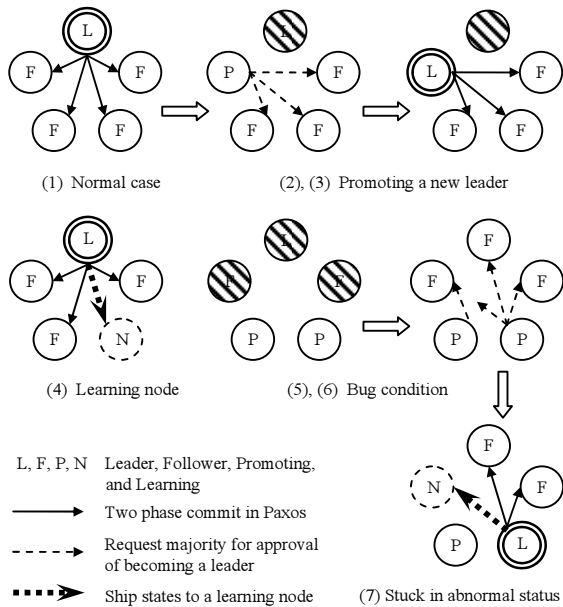


Figure 6: Leader election in Paxos implementation and the snapshots of states that lead to the bug.

with failure cases is important. Therefore we inject a failure module along with the state exposor library, which can fail the MPS process.

An important invariant is *safety and liveness of agreement*: all nodes execute identical sequence of commands with sequence number always increasing by 1 (safety), and the system can make progress as long as there exist majority of nodes (liveness). To check this invariant, D³S exposes every executed command in each node with its sequence number. After days of running, D³S detected no violations. We did find a careless pointer error in a rare code path, thanks to failure injection.

Then we turned our attention to the leader election implementation of MPS—this part of the service was less well specified and proven by the development team, compared to their implementation of the Paxos protocol. A node can be in one of the four status: Leader, Follower, Promoting and Learning. In normal case, one Leader broadcasts incoming commands to Followers using the Paxos protocol (Figure 6.(1)). Promoting and Learning are only transient status. Promoting is used when leader is absent, and the promoting node needs to get majority’s approval to become the new leader (Figure 6.(2) and (3)). A Learning node is out-of-date in execution of commands, and is actively transferring state from another node (Figure 6.(4)). Leader election should make the system always go back to the normal case after failures. Therefore, we wrote a predicate to check the status of nodes and catch persistent deviation from the normal case.

After running for hours with randomly injected failures, this predicate detected a persistent abnormal status, as shown in (5) ~ (7). We name the five nodes as A ~ E in counterclockwise order. After A, B, E crash and recover, both C and D are promoting themselves (in (5)). Only D gets majority’s approval (A, E and itself). C is approved by B but the messages to A and E are lost (in (6)). Later when D’s promoting request arrives at B, B happens to be out-of-date compared with D, so B switches to Learning (in (7)). By design, C will become Follower when it learns the new leader. But now C considers A, E are dead and only keeps requesting B, which is a learning node and will just ignore any requests except state transfers. Therefore, C can be stuck in Promoting for a long time. Although this bug does not corrupt the Paxos properties right away, it silently excludes C from the group and makes the system vulnerable to future failures. This bug was confirmed by people working on the code.

This is a fairly complicated bug. Although the system has correct external behavior, D³S helps us discover the bug with always-on checking of internal states. This bug had been missed by log verification. We guess the reason might be that, when the abnormality shows up for a short time during a long execution, it is hard to detect bugs with postmortem analysis. Another lesson is that less rigorously specified components (e.g., leader election) are harder to get right. Therefore, it is productive for testers to write and install “ad-hoc” predicates on-the-fly to probe the suspected components, as D³S enables.

5.3 Web search engine

Another research group in our lab has developed an experimental search engine that powers an online vertical search [37]. It has a frontend server which distributes queries to a set of backend index servers and collects results. The index is partitioned among the index servers using the partition-by-document scheme. Each index server further partitions its responsible index into five tiers, according to document popularity. Only the first tier is kept in memory. Given a query of terms, an index server first retrieves the documents from the first tier. If there are 100 hits, the index server returns, otherwise it tries the next tier, and so on. The system is deployed in 250 servers, each having 8 GB memory and 1 TB disk. We use a dedicated server to run as a verifier, which communicates with all index servers and the frontend. The frontend keeps the index servers by replaying query logs from a commercial search engine.

Developers of the system were not satisfied about the performance. They had statistics on total latency for queries, but they wanted to know what contributed to the latency of the critical path, and how much time was spent

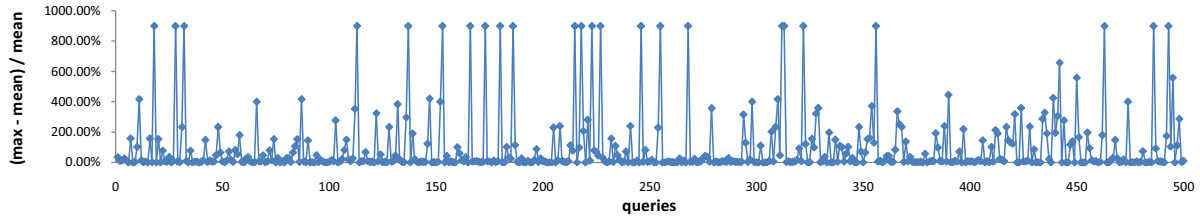


Figure 7: Load imbalance in the web search engine.

on each tier. We used D³S to interpose on every function in the critical path (e.g., fetching documents from each tier, intersecting lists of terms, and sorting the results), and exposed the execution time and the related terms in the query. This provided detailed performance data. We wrote predicates to probe different aspects of the latency to catch abnormal queries.

One predicate we wrote checked load balance. For each index tier, the predicate calculates $Pr = (max - mean) / mean$, using maximum and mean of the latency in the tier among index servers. Larger Pr means less balance of the load. Since the frontend needs to collect results from all the index servers, unbalanced load may result in a performance degradation, since the slowest index server determines the final response time.

Figure 7 shows the values of Pr of 500 selected queries over 10 index servers. For the queries whose Pr is greater than one (i.e., $max > 2 \times mean$), there are always one or two index servers running much slower than others. Further diving into the latency measures with other predicates (e.g., the number of visited tiers in different index servers), we found that simple hashing does not balance the load across the index server uniformly, and the imbalance degrades performance significantly when the query has more than one terms.

D³S is useful in terms of flexibility and transparency. Adding logs to the search engine code and performing postmortem analysis of these logs will yield the same insights. However, it is more work and doesn't allow refining predicates on-the-fly.

5.4 Chord

DHTs are by now a well-known concept. There are availability measures of DHTs, in terms of convergence time and lookup success rate in case of failures. However, the routing consistency in DHTs is less well understood. Routing consistency means that a lookup returns the *correct* node holding the key, which may not happen when nodes fail or the network partitions [11]. For example, a DHT put for a key may store the key at a node that after healing of the network isn't the successor or isn't even in the successor list, and the key becomes unavailable. Similarly, it may happen that several nodes think

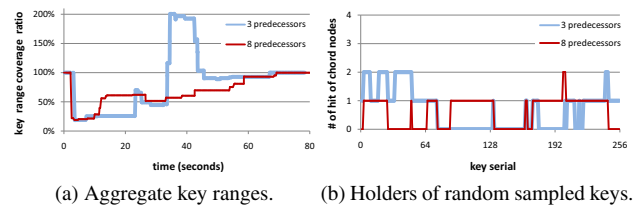


Figure 8: *i3*-Chord experimental results.

they are *the* successor of a key, which could lead to inconsistencies. Because availability and consistency are tradeoffs in distributed systems [17], a better study on DHTs should measure these two issues altogether.

We ran an experiment with 85 Chord nodes on 7 servers. We ran the *i3* [1] Chord implementation, which we ported to Windows. We used the predicate in section 2 (Figure 3) to check the aggregate key ranges held by the nodes, with 4 verifiers for the first stage and 1 verifier for the final stage. This predicate checks a necessary but insufficient condition for the integrity of the DHT logical space. If the output value is below 100%, then the system must have “holes” (i.e., no node in the system believes it is the successor of a key range). This indicates unavailability. On the other hand, an output value above 100% indicates overlaps (i.e., several nodes believe they are the successor).

In *i3*, a Chord node maintains several predecessors and successors, and periodically pings them for stabilizing the ring. We used two configurations for the number of predecessors, one is 3 and the other is 8. The number of successors is set to the same value as the number of predecessors in either configuration. We crashed 70% of nodes and then monitored the change of the aggregated key ranges.

Figure 8a shows the results of the two configurations. The 3-predecessor case has unpredictable behavior with respect to both consistency and availability. The total key range value oscillates around 100% before converging. The 8-predecessor case does better, since it never exceeds 100%. The swing behavior in the 3-predecessor case results from nodes that lose their predecessors and successors. Such nodes use the finger table to rejoin

the ring. This case is likely to result in overlapping key ranges. With more predecessors and successors, this case happens rarely.

With the total key range metric, the “holes” and “overlaps” can offset each other, and therefore when the metric is below 100%, there could be overlaps as well. To observe such offsets, we added a predicate to sample 256 random points in the key space, and observed the number of nodes holding these points. Figure 8b shows a snapshot taken at the 25th second. Despite that the total key range never exceeds 100% for the 8-predecessor configuration, overlaps occur (see key serial number 200), which indicate inconsistency. The figure also shows a snapshot when both inconsistency and unavailability occur in the 3-predecessor configuration.

D³S allows us to quantitatively measure the tradeoffs between availability and consistency in a scalable manner using D³S’s support for a tree of verifiers and for sampling, and reveals the system behavior at real time. Although we did these experiments in a test-lab, repeating them in a wide-area large-scale deployment is practical. For a wide-area online monitoring system, D³S’s scalable and fault-tolerant design should be important.

5.5 BitTorrent client

We applied D³S to libtorrent [2] release 0.12, with 57 peers running on 8 machines. The peers download a file of 52 MB. The upload bandwidth limit varies from 10 KB/s to 200 KB/s in different experiments, and peers finish downloading in 15 minutes to 2 hours. The bugs we found (and acknowledged by the developers) illustrate the use of real time monitoring with D³S predicates.

Similar to Chord, BitTorrent exhibits complex behavior which is difficult to observe in detail. We start with investigating key properties of the running system with predicates in order to gain some insights. The first property we looked at was the peer graph, because the structure of graph is important to dissemination of data.

We exposed the neighbor list from every peer, and used a two-stage predicate to form an aggregation tree, similar to the total key range predicate in Chord. At the final stage, we obtained the entire graph in real time. We printed the graph to a console and kept refreshing it, in order to observe the graph changes. we found an abnormality right away: sometimes a peer had more than 300 neighbors (recall that we used only 57 peers). We inspected the code and found that, when adding a new IP address into the neighbor list, libtorrent did not check if the same IP address already existed. This resulted in duplicated IP addresses in neighbor list, and degraded performance because peers may had less choices when downloading from neighbors. We fixed the bug and continued with subsequent experiments.

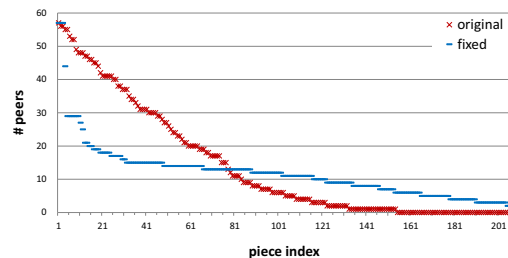


Figure 9: Piece distributions over peers when finishing 30% downloading.

After the peer graph looked correct, we started another predicate to compute how pieces are distributed over peers. Every time when a peer downloads a piece, the predicate exposes the piece number along with its own ID. The predicate aggregates the vector of available pieces in every peer, again using a two stage aggregating tree. With simple visualization (i.e., printing out the distribution of pieces at real time), we observed that some pieces spread much faster than the others. Figure 9 illustrates a snapshot of the numbers of pieces over peers. Some pieces are distributed on all peers in a short time, while other pieces make less progress. This was not what we expected: pieces should have been selected randomly and therefore the progresses should *not* have differed that dramatically. Thus, we suspected that peers are converging on same pieces. We examined the code and found that the implementation *deterministically* chose which pieces to download: for pieces with the same number of replicas, all clients chose replicas in the same order. We fixed the bug and collected the data again. As shown in Figure 9, the progress of pieces is much closer to random selection.

After fixing these bugs, we implemented algorithms that detect free riders who disproportionately download compared to upload. Our purpose was to further expand the use of D³S from debugging-focused scenarios to online monitoring. Of the 57 peers, the upload bandwidths of Peer 46~56 were limited to 20 KB/s, while other peers’ upload bandwidths remained 200 KB/s. All peers had unlimited download bandwidth. Peer 46~56 were more likely to be free riders, compared to the other peers [33]. We compute contributions of the peers using predicates that implement the EigenTrust algorithm [22]. The predicates collect the source of every downloaded piece in each peer, and calculate EigenTrust in a central server. As shown in Figure 10, the predicates successfully distinguished peers via their contributions.

5.6 Summary of debugging experience

Using examples we have shown that D³S helps find non-trivial bugs in deployed, running systems. The effective-

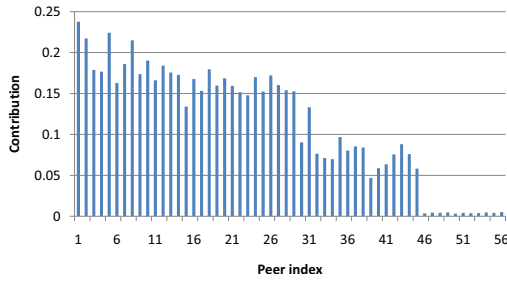


Figure 10: The contributions of peers (free riders are 46~56).

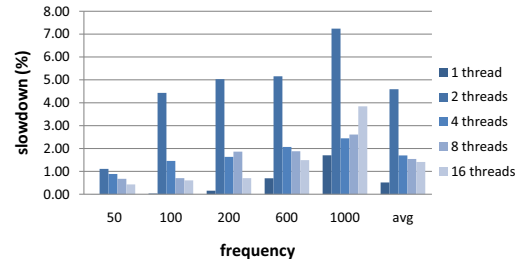
ness of D³S depends on whether or not we have useful predicates to check. When a system already has specifications and invariants (e.g., at the component level), which is common for complex, well designed systems, D³S is effective, because the predicates can check the invariants. Writing the predicates is mostly an easy task for developers, because they are allowed to use sequential programs on global snapshots. When a system doesn't have a clear specification (e.g., in performance debugging), D³S is more like a dynamic log-collecting and processing tool, which can help zooming into specific state without stopping the system. This helps developers probing the system quickly, and eventually identify useful predicates.

D³S is not a panacea. Component-level predicates are effective for debugging a single system with a good specification. However, when debugging large-scale web applications running in data centers, this approach is sometimes insufficient. First, data center applications often involve a number of collaborative systems that interact with each other. When unexpected interactions happen that lead to problems (e.g., performance degradation), developers have little information about which system they should inspect for the problem. Second, these systems evolve on daily basis, and sometimes there are no up-to-date specifications to check. These issues are what our on-going research on D³S aims to address.

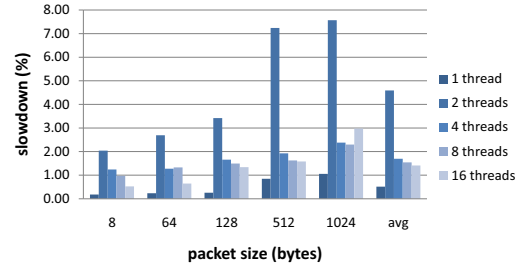
6 Performance Evaluation

This section studies the performance of D³S, using the machine configuration described at the beginning of Section 5.

We first evaluate overhead of checking on a running system. This overhead is caused by the cost of exposing state, and depends on two factors: the frequency of exposing state and the average size of the state exposed. To test the overhead under different conditions, we use a micro benchmark in which the checked process starts various number of threads. Each thread does intensive computation to push CPU utilization close to 100%. Figure 11 shows the overhead. We can see that the state ex-



(a) Slowdown with average packet size 390 bytes and different exposing frequencies.



(b) Slowdown with average frequency 347 /s and different exposing packet sizes.

Figure 11: Performance overhead on system being checked.

poser is lightweight and in general the overhead is around 2%. The largest overhead happens when the process has 2 threads of its own, which maps perfectly to the dual-core CPU. State exposer brings one additional thread, and thus increases the thread scheduling overhead. In this case the overhead is still less than 8%.

These results are consistent with all the systems checked. Systems that are neither I/O nor CPU intensive (e.g., Chord and Paxos) have negligible overhead; BitTorrent and Web search have less than (< 2%) overhead. The impact to PacificA varies according to system load (Figure 12). We created 100 slices and we vary the number of concurrent clients, each sends 1000 random reads and writes per second with average size 32KB per second. The overhead is less than 8%. A PacificA machine generates in average 1,500 snapshots per second, and consumes at the peak time less than 1000 KB/s additional bandwidth for exposing states to verifier. On average, exposing states uses less than 0.5% of the total I/O consumption. These results encourage adopting D³S as an always-on facility.

Second, we evaluate the impact on performance of PacificA when we start new predicates. We start checking all predicates in Section 5.1 at the 60th second. Before that there is no state exposer injected to PacificA. Figure 13 shows the total throughput seen by clients. Given that PacificA itself has fluctuating throughput due to reorganizing disk layout (see [26]), there is no visible impact on performance when starting new predicates.

In addition, we evaluate the failure handling of D³S.

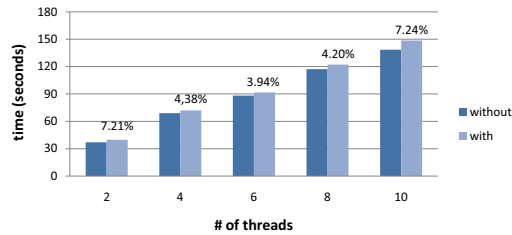


Figure 12: Performance overhead on PacificA with different throughput.

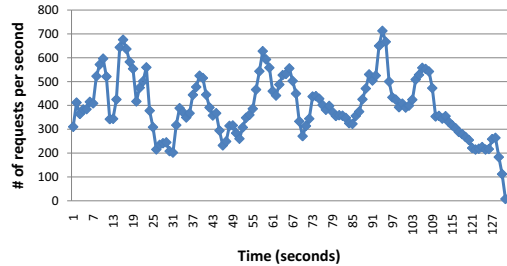


Figure 13: Throughput of PacificA when a predicate starts.

In the above PacificA testing, we start 3 verifiers and kill one at the 30th second. After the failure is detected, the uncovered key range are repartitioned. Figure 14 shows how the load of the failed verifier is taken over by the remaining verifiers. The fluctuation reflects the nature of PacificA, which periodically swaps bulk data between memory and disk.

7 Related Work

Replay-based predicate checking. People have proposed to check replayed instances of a distributed system for detecting non-trivial bugs that appear only when the system is deployed [27, 16]. D³S addresses one critical weakness in that replaying the entire execution of a large-scale system is prohibitively expensive. The advantage that replay brings is to repeatedly reproduce the execution to aid debugging, and the role of the online checking is to accurately position and scope the replay once a bug site is reported. We see replay as a key complementary technology to online predicate checking. The ultimate vision is to use online checking to catch violations, and then enable time-travel debug of a recent history with bounded-time replay.

Online monitoring. P2 monitor [34] is designed for online monitoring of distributed properties, which is close to ours in spirit. However, D³S differs in a number of ways. First, D³S allows expressing predicates on global snapshots of states, while P2 monitor requires its user to take care of collecting and ordering the monitored states. Second, D³S works on legacy systems, while P2 monitor

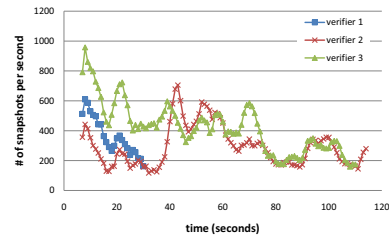


Figure 14: Load when verifier 1 fails at 30th second.

is confined to systems built with OverLog language [28]. Finally, D³S can tolerate failures from both the system being checked and itself.

Log analysis. There is a collection of literature that relies on logs for postmortem analysis, including analyzing statistics of resource usage [3], correlating events [5], tracking dependency among components [10, 14] and checking causal paths [32]. Using a specific and compact format, the work in [36] can scale event logging to the order of thousands of nodes. Fundamentally, logging and online predicate checking all impose runtime overhead to expose information for analysis. D³S is, at a minimum, an intelligent logger that collects states in a scalable and fault-tolerant way. However, the D³S's advantage in exposing new states on-the-fly and allowing a simple programming model can significantly improve debugging productivity of developers in practice.

Large-scale parallel applications. Parallel applications generally consist of many identical processes collectively for a single computation. Existing tools can check thousands of such processes at runtime and detect certain kinds of abnormalities, for example, by comparing stack traces among the processes [4, 30], or checking message logs [13]. In contrast, D³S works for both identical and heterogeneous processes, and is a general-purpose predicate checker.

Model checking. Model checkers [23, 31, 38] virtualize the environments to systematically explore the system space to spot a bug site. The problem of state explosion often limits the testing scale to small systems compared to the size of deployed system. Similarly, the testing environment is also virtual, making it hard to identify performance bugs, which require a realistic environment and load. D³S addresses the two problems by checking the deployed system directly.

8 Conclusion and Future Work

Debugging and testing large-scale distributed systems is a challenge. This paper presented a tool to make debugging and testing of such systems easier. D³S is a flexible and versatile online predicate checker that has shown its promise by detecting non-trivial correctness and perfor-

mance bugs in running systems.

As future work, we are exploring several directions. We are pushing forward our vision of combining online predicate checking and offline replay. We are also exploring tools to debug data center applications that are composed of many systems so that a developer can easily find bugs due to unexpected interactions between the systems.

Acknowledgments

We would like to thank our shepherd, Amin Vahdat, and the anonymous reviewers for their insightful comments. Thanks to our colleagues Haohui Mai, Zhilei Xu, Yuan Yu, Lidong Zhou, and Lintao Zhang for valuable feedback.

References

- [1] i3 implementation: <http://i3.cs.berkeley.edu/>.
- [2] libtorrent: <http://libtorrent.sourceforge.net/>.
- [3] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).
- [4] ARNOLD, D. C., AHN, D. H., DE SUPINSKI, B. R., LEE, G., MILLER, B. P., AND SCHULZ, M. Stack trace analysis for large scale debugging. In *IPDPS* (2007).
- [5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004).
- [6] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *OSDI* (2006).
- [7] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *PODC* (2007).
- [8] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. In *JACM* (1996).
- [9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).
- [10] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic, internet services. In *DSN* (2002).
- [11] CHEN, W., AND LIU, X. Enforcing routing consistency in structured peertopeer overlays: Should we and could we? In *IPTPS* (2006).
- [12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).
- [13] DESOUSA, J., KUHN, B., AND DE SUPINSKI, B. R. Automated, scalable debugging of MPI programs with Intel message checker. In *SE-HPCS* (2005).
- [14] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-Trace: A pervasive network tracing framework. In *NSDI* (2007).
- [15] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX* (2006).
- [16] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOEY, T., AND STOICAZ, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).
- [17] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT. ACM* 33, 2 (2002).
- [18] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989).
- [19] GUO, Z., WANG, X., LIU, X., LIN, W., AND ZHANG, Z. BOX: Icing the APIs. MSR-TR-2008-03.
- [20] HUNT, G., AND BRUBACHER, D. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium* (1999).
- [21] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).
- [22] KAMVAR, S. D., SCHLOSSER, M. T., AND GARCIA-MOLINA, H. The EigenTrust algorithm for reputation management in P2P networks. In *WWW* (2003).
- [23] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).
- [24] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978).
- [25] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
- [26] LIN, W., YANG, M., ZHANG, L., AND ZHOU, L. PacificA: Replication in log-based distributed storage systems. MSR-TR-2008-25.
- [27] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. WiDS checker: Combating bugs in distributed systems. In *NSDI* (2007).
- [28] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *SOSP* (2005).
- [29] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004).
- [30] MIRGORODSKIY, A. V., MARUYAMA, N., AND MILLER, B. P. Problem diagnosis in large-scale computing environments. In *SC* (2006).
- [31] MUSUVATHI, M., AND ENGLER, D. Model checking large network protocol implementations. In *NSDI* (2004).
- [32] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).
- [33] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. A measurement study of peer-to-peer file sharing systems. In *MMCN* (2002).
- [34] SINGH, A., ROSCOE, T., MANIATIS, P., AND DRUSCHEL, P. Using queries for distributed monitoring and forensics. In *EuroSys* (2006).
- [35] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In *Sigcomm* (2002).
- [36] VERBOWSKI, C., KICIMAN, E., KUMAR, A., DANIELS, B., LU, S., LEE, J., WANG, Y.-M., AND ROUSSEV, R. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI* (2006).
- [37] WEN, J.-R., AND MA, W.-Y. Webstudio: building infrastructure for web data management. In *SIGMOD* (2007).
- [38] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *OSDI* (2004).

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *login*., the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patrons

Google
Microsoft Research
NetApp

USENIX Benefactors

Hewlett-Packard
Linux Pro Magazine

USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
FOTO SEARCH Stock Footage
and Stock Photography
Raytheon

rTIN Aps

Splunk
Taos
Tellme Networks
Zenoss

USENIX Partners

Cambridge Computer Services,
Inc.
cPacket Networks
EAGLE Software, Inc.
GroundWork Open Source
Solutions
Hyperic

IBM

Infosys
Intel
Interhack
Oracle
Ripe NCC
Sendmail, Inc.
Sun Microsystems, Inc.
UUNET Technologies, Inc.
VMware

SAGE Partner

MSB Associates

ISBN-13: 978-1-931971-58-4

90000



9 781931 971584